Maven:

When we create a Test java file in test directory we need to make sure that our **class** name includes ***Test** word

Note: Make sure you do not name your class simply Test, it will conflict with Test annotation From TESTNG

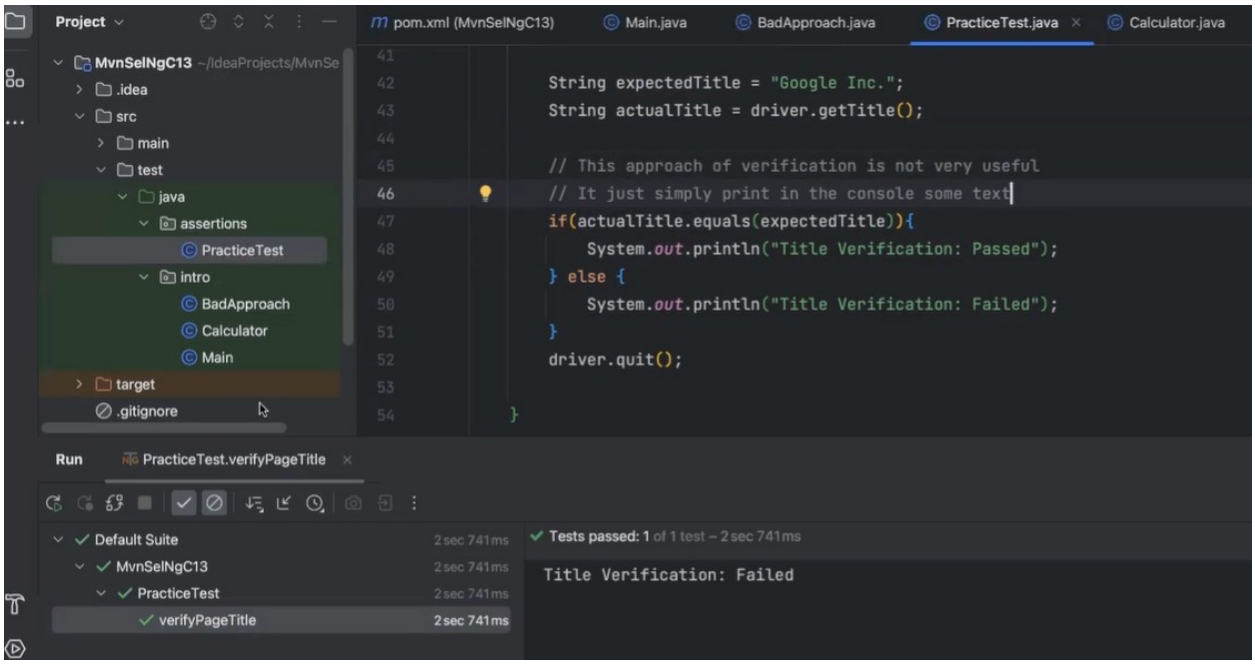
```
// In testNG first thing we get rid of main method
// We will write each test as a separate method
// The method can have any name you want
// Usually we provide a description what we test
// Usually the test methods are void
```

FIRST APPROACH : NOT USEFUL IF IT HAS A CORRECT CODING

```
package assertions;

import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class PracticeTest {

	@Test
	public void verifyLogoIsDisplayed() {
		WebDriver driver = new ChromeDriver();
		driver.get("https://www.google.com/");

		// Correct XPath for finding the logo element on Google's homepage
		WebElement logo = driver.findElement(By.xpath("//img[@id='hplogo']"));

		// Check if the logo is displayed
		boolean logoIsDisplayed = logo.isDisplayed();

		// Verify if the logo is displayed and print the result
		if (logoIsDisplayed) {
			System.out.println("Logo Verification: Passed");
		} else {
			System.out.println("Logo Verification: Failed");
		}

		driver.quit();
	}

	@Test
	public void verifyPageTitle() {
		WebDriver driver = new ChromeDriver();
		driver.get("https://www.google.com/");

		String expectedTitle = "Google";
```

```
            String actualTitle = driver.getTitle();

            // Verify if the title is correct and print the result
            if (actualTitle.equals(expectedTitle)) {
                  System.out.println("Title Verification: Passed");
            } else {
                  System.out.println("Title Verification: Failed");
            }

            driver.quit();

      }
}
```

IF THE TITLE VERIFICATION IS FAILED, THE TEST IS PASSED ! WE NEED ANOTHER APPROACH.



Instead we will use Assert class from testNG

TestNG provides you with a lot methods to test conditions, values etc...

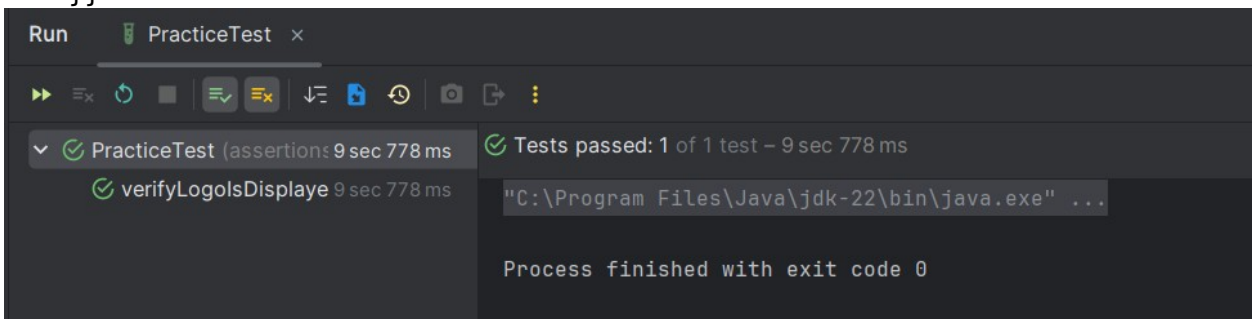
In fact they will notify the framework that a test failed or passed with an exception

We will provide to ***assertTrue()*** a boalean condition

Second Approach: with *assertTrue()*

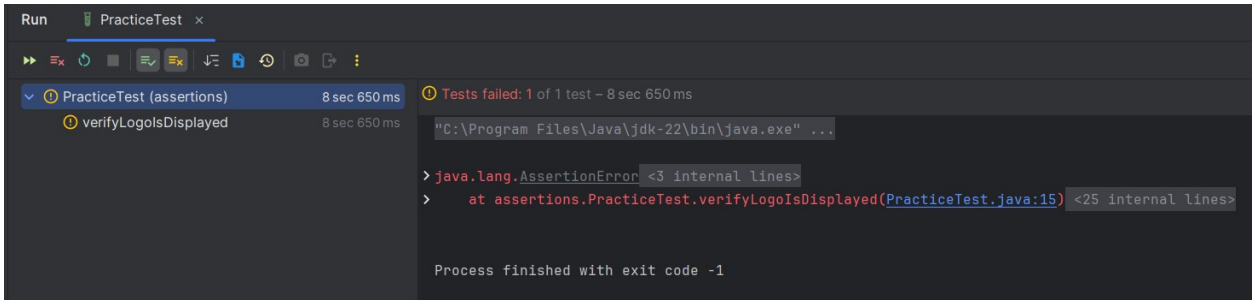
Result is TRUE :

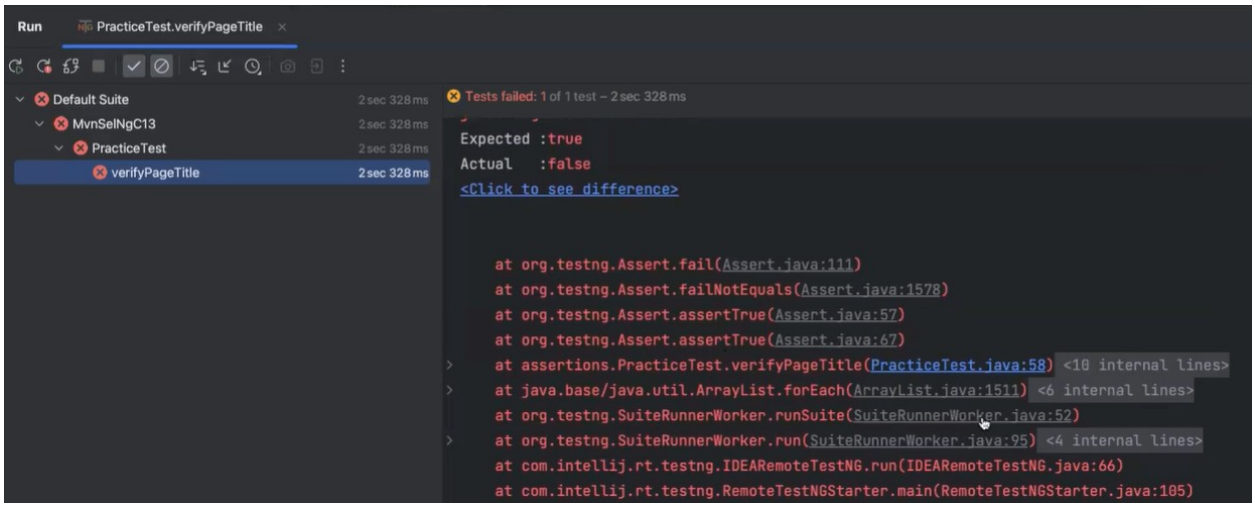
```java
package assertions;
import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PracticeTest {

    @Test
    public void verifyLogoIsDisplayed() {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.google.com/");
        String expectedTitle = "Google";
        String actualTitle = driver.getTitle();
        Assert.assertTrue(actualTitle.equals(expectedTitle));
        driver.quit();
    }}
```


Run
PracticeTest
PracticeTest (assertions 9 sec 778 ms
verifyLogoIsDisplaye 9 sec 778 ms
Tests passed: 1 of 1 test – 9 sec 778 ms
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
Process finished with exit code 0

Result is FALSE :

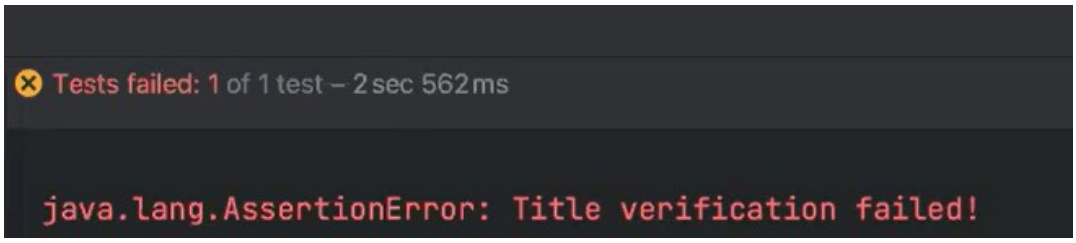
```java
package assertions;
import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PracticeTest {

    @Test
    public void verifyLogoIsDisplayed() {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.google.com/");
        String expectedTitle = "Google Inc";
        String actualTitle = driver.getTitle();
        Assert.assertTrue(actualTitle.equals(expectedTitle));
        driver.quit();
    }}
```





Adding an Alert when existing a failure:

Assert.*assertEquals*("Title Verification: Failed", actualTitle, expectedTitle);

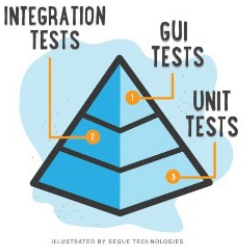


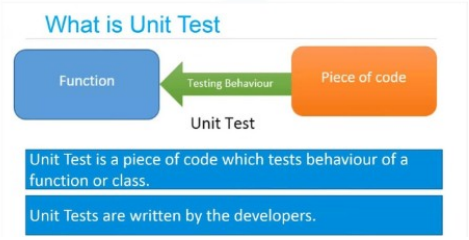
- ***The assert methods will notify the project if test failed or passed***
- ***This Assert class is also called Hard Assert***
- ***If an assertion fails, the test case get marked as failed and it doesn't run the subsequent code.***

Assert.assertTrue(actualTitle.equals(expectedTitle), message: "Title verification failed!");

Unit Testing

- Unit Testing is used to verify a small chunk of code.
- It is performed by the developers.
  - Both Front End and Back End developers do unit test for their code.
  - Developers write some test code to test their block of codes, methods etc.
  - Unit tests are much **faster** than UI Testing.
  - Unit Testing is used to identify defects **early** in software development cycle.
  - It helps developer to write more readable, reliable and bug-free code.
- Unit Testing is NOT performed by Automation Testers.




What is Unit Test
Function
Testing Behaviour
Piece of code
Unit Test
Unit Test is a piece of code which tests behaviour of a function or class.
Unit Tests are written by the developers.

TestNG

Testing is about comparing **Expected Result** vs **Actual Result**
  - To achieve this goal, we have to use some sort of testing framework.

TestNG is one of the most popular **unit testing framework** to test Java applications.
  - Framework are 2 kinds: **Built in frameworks** (such as TestNG, JUnit etc.) & **User Defined**.

TestNG is mainly used for **Unit testing** of Java projects
However, it can be used with Selenium Webdriver to automate testing of Web applications.

With TestNG, you can **assert expectations**, which allows you to PASS or FAIL a test case based on the actual result.

TestNG

- **TestNG**
  - NG stands for "**Next Generation**"
  - It shows how many test cases are passed, failed and skipped.
  - Note: if exceptions happen in TestNG, only that method fails, rest of the methods continue to run.
  - You can execute failed test case separately. *(Can't achieve it with main method.)*

- TestNG offers below important features:
  - Order of execution→ @Test(priority=1)
  - Parallel Execution → by using XML file. You can do cross browser-testing in parallel.
  - Parameterized Tests → by using **@Data Providers**
  - Using TestNG you get an html report by default
  - Dependency Test→ allows a test method to depend on a single or a group of test methods.
  - Uncaught exceptions are automatically handled by TestNG without terminating the test prematurely. These exceptions are reported as failed steps in the report.

Setup

- To install TestNG to our Maven project, we just have to add below dependency:

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.4.0</version>
    <scope>test</scope>
</dependency>
```

ASSERTIONS

- `assertTrue(boolean condition, "reason why failed");`
  - This method verifies whether given parameter results in true.
  - assertTrue (3 < 4);
- `assertFalse(boolean condition, "reason why failed");`
  - This method verifies whether given parameter results in false.
  - assertFalse(3 < 4);
- `fail("custom message");`
  - This method fails the current test case without checking for any conditions
  - It is useful in case where you have to fail a test case **when an particular exception occurs.**

assertFalse

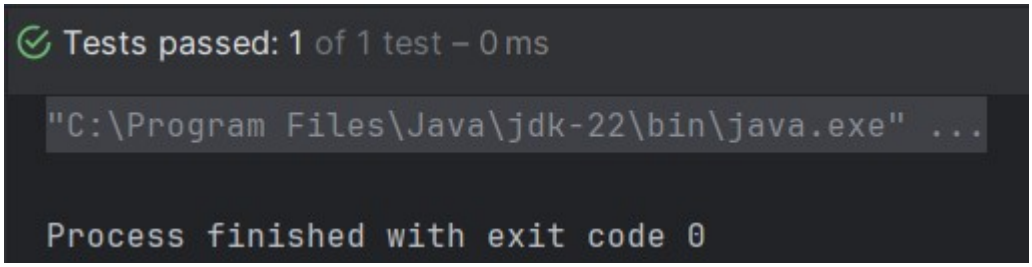
```
package assertions;

import org.junit.Assert;
import org.junit.Test;

public class AssertionPracticeTest {

	/*
	 * Assert.assertTrue(boolean condition, <Optional> String message); We can use
	 * as argument Everything that results to a boolean value It can be something
	 * like (4<5) Or boolean result = 4<5; -> use result as a condition
	 *
	 */
	@Test
	public void testEConditionTrue() {
		int myAge = 30;
		boolean guestCanEnter = myAge >= 18; // true
		Assert.assertTrue("You cannot enter. Your age is lower that expected." +
myAge, guestCanEnter);
		String hasTicket = "yes";
		Assert.assertTrue("You should have a ticket in order to get it",
hasTicket.equalsIgnoreCase("no"));

	}

	@Test
	public void testEConditionFalse() {
		// If we want to check that a button is disabled how to do it?
		// button.isEnabled();
		boolean buttonIsDisabled = false;
		Assert.assertFalse("The button is enabled!", buttonIsDisabled);
	}
}
```

assertTrue

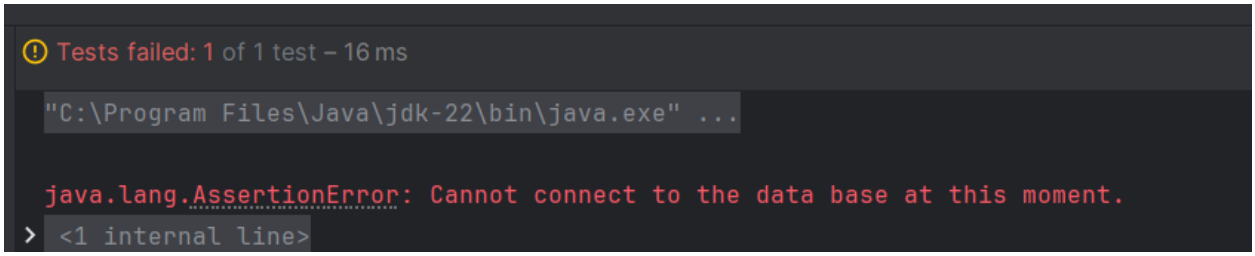
```
package assertions;

import org.junit.Assert;
import org.junit.Test;

public class AssertionPracticeTest {

	/*
	 * Assert.assertTrue(boolean condition, <Optional> String message); We can use
	 * as argument Everything that results to a boolean value It can be something
	 * like (4<5) Or boolean result = 4<5; -> use result as a condition
	 *
	 */
// Test 1 we expect a true result:

	@Test
	public void testEConditionTrue() {
		int myAge = 30;
		boolean guestCanEnter = myAge >= 18; // true
		Assert.assertTrue("You cannot enter. Your age is lower that expected." +
myAge, guestCanEnter);
		String hasTicket = "yes";
		Assert.assertTrue("You should have a ticket in order to get it",
hasTicket.equalsIgnoreCase("no"));

	}
// Test 2 we expect a false result:

	@Test
	public void testEConditionFalse() {
		// If we want to check that a button is disabled how to do it?
		// button.isEnabled();
		boolean buttonIsDisabled = false;
		Assert.assertFalse("The button is enabled!", buttonIsDisabled);
	}

	/*
	 * Assert.fail() this will intentionally fail the test case
	 **/
	// FAIL() Assertion
	// Case 1
	// If we want to fail an assertion like disconnect a Database if it is connected
	@Test
	public void testDatabaseConnectionISConnected() {
		boolean isConnected = true;
		if (!isConnected)
			Assert.fail("Cannot connect to the data base at this moment.");
	}

	// Case 2
	// If we want to fail an assertion like disconnect a Database if it is connected
	@Test
	public void testDatabaseConnectionNOTConnected() {
```

```
            boolean isConnected = false;
            if (!isConnected)
                  Assert.fail("Cannot connect to the data base at this moment.");
      }
}
```

CASE1



CASE2:



TASK

- **Write a method**

Navigate to Amazon homepage.

IVerify logo is displayed with assertTrue() method

Run your code with @Test annotation

- **Write a method**

Navigate to Amazon homepage

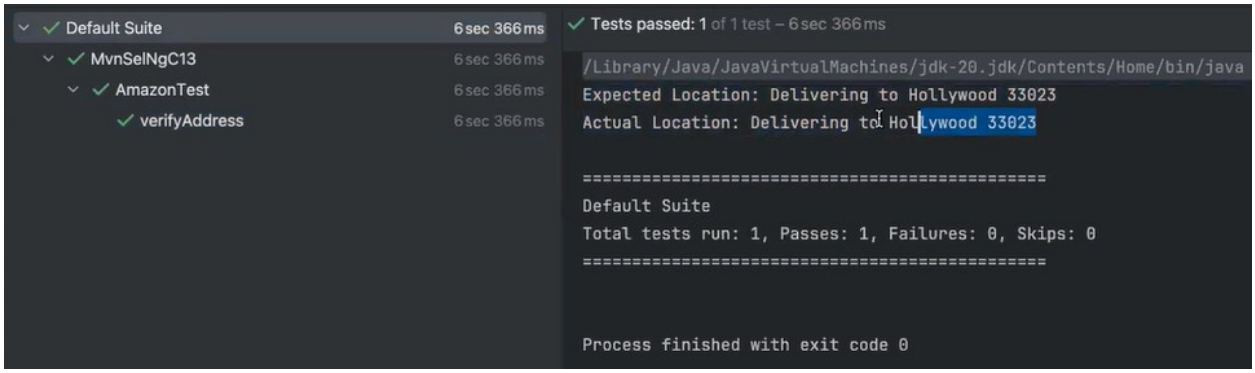
Get the City name [by default it is: Select your address]

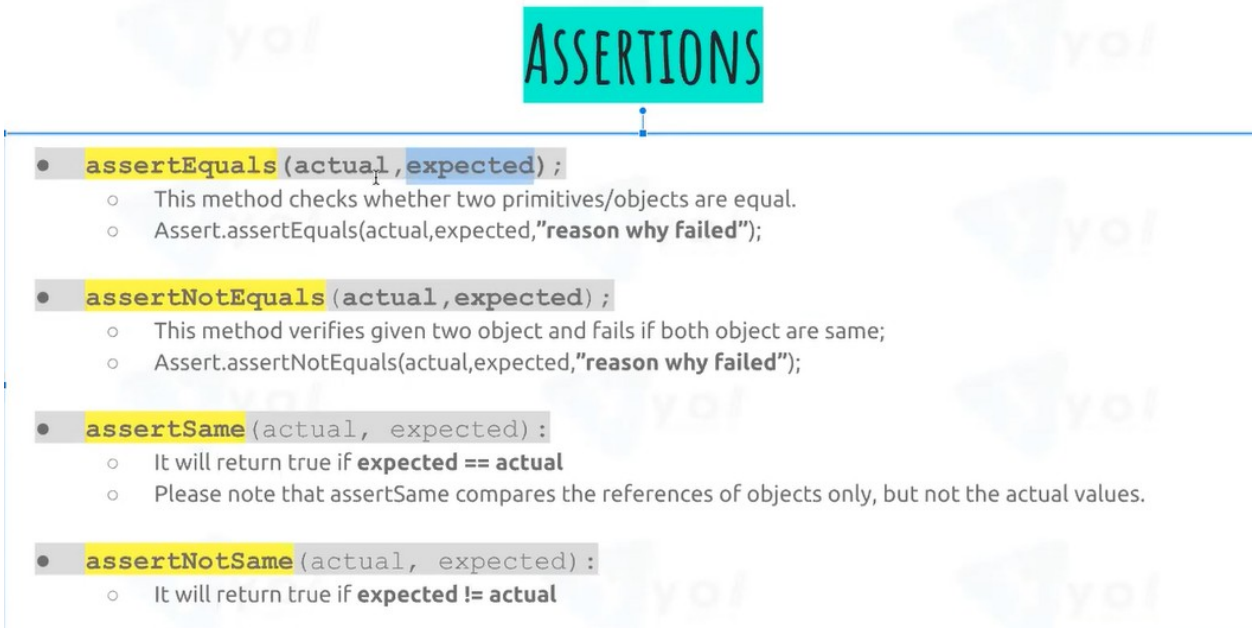
Fail the test if it is not equal to: Select your address

```
package assertions;

import org.junit.Assert;
import org.junit.Test;
```

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;


    public class AmazonTest {

        @Test
        public void verifyLogoIsDisplayed() {
            WebDriver driver = new ChromeDriver();
            driver.get("https://www.amazon.com/");
            WebElement logo = driver.findElement(By.id("nav-logo-sprites"));
            Assert.assertTrue("Logo verification failed.", logo.isDisplayed());
        }
    }


package assertions;

import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class AmazonTest {
      /*
       * Write a method Navigate to Amazon homepage Verify logo is displayed with
       * assertTrue() method Run your code with @Test annotation
       */

      @Test
      public void verifyLogoIsDisplayed() {
            WebDriver driver = new ChromeDriver();
            driver.get("https://www.amazon.com/");
            WebElement logo = driver.findElement(By.id("nav-logo-sprites"));
            Assert.assertTrue("Logo verification failed.", logo.isDisplayed());
      }

      /*
       * Given user navigates to "https://www.amzon.com" When user lands on home page
       * Then user is able to see the Update Address And the address should be
       * "Your Address"
       */
      @Test
      public void verifyAddress() {
            WebDriver driver = new ChromeDriver();
            driver.get("https://www.amazon.com/");
// Locate the city placeholder
// glow-ingress-line2
            WebElement addressLabel = driver.findElement(By.id("glow-ingress-line2"));
            boolean cityPlaceholderIsDisplayed = addressLabel.isDisplayed();
```

```
            Assert.assertTrue("City label verification failed!",
cityPlaceholderIsDisplayed);
            String expectedLocation = "Delivering to Hollywood 33023";
            WebElement addressLocation = driver.findElement(By.id("glow-ingress-line1"));
            String actualLocation = addressLocation.getText();

            System.out.println("Expected Location: " + expectedLocation);
            System.out.println("Actual Location: " + expectedLocation);
            Assert.assertTrue("Delivery address verification failed.",
expectedLocation.equals(actualLocation));
            driver.quit();
      }
}
```


Default Suite 6 sec 366 ms
MvnSelNgC13 6 sec 366 ms
AmazonTest 6 sec 366 ms
verifyAddress 6 sec 366 ms
Tests passed: 1 of 1 test – 6 sec 366 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java
Expected Location: Delivering to Hollywood 33023
Actual Location: Delivering to Hollywood 33023
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
Process finished with exit code 0


ASSERTIONS
assertEquals(actual,expected);
This method checks whether two primitives/objects are equal.
Assert.assertEquals(actual,expected,"reason why failed");
assertNotEquals(actual,expected);
This method verifies given two object and fails if both object are same;
Assert.assertNotEquals(actual,expected,"reason why failed");
assertSame(actual, expected):
It will return true if expected == actual
Please note that assertSame compares the references of objects only, but not the actual values.
assertNotSame(actual, expected):
It will return true if expected != actual

assertEquals

```
      /*
       * Assert.assertEquals() will take two arguments of any data type (primitve
```

```
	 * non-primitive) It will compare their values - same as equals() for strings
	 **/
	@Test
	public void testEquality() {
		Assert.assertEquals(10, 10, "Values are not equal");
		Assert.assertEquals("Hello", "Hello", "Values are not matching");
		String str = "Student";
		String str2 = new String("Student");
		Assert.assertEquals(str, str2, "Values are not matching.");
	}
}
```

The test will pass because str1 and str2 are equal when using `assertEquals`.

The difference between String str = "Student"; and String str2 = new String("Student"); lies in how the two String objects are created and stored in memory.

1. String str = "Student";

- **String Literal:** This creates a String object in the **String pool** (a special area in the Java heap memory).
- **Optimization:** When you create a String using a literal, the JVM checks the String pool to see if an identical string already exists. If it does, it will reuse the existing string instead of creating a new one.
- **Memory Efficient:** No new object is created if a matching string is found in the pool, making it more memory efficient.

2. String str2 = new String("Student");

- **Explicit Object Creation:** This forces the creation of a new String object in the **heap memory**, even if an identical string already exists in the String pool.
- **Not Reusing:** The new keyword explicitly instructs Java to create a new object in memory, bypassing the String pool.

Code Example with Assertion:

String str = "Student";

String str2 = new String("Student");

// Assert that the values are equal (this compares content, not memory reference)

Assert.assertEquals(str, str2, "Values are not matching.");

What Happens in the Assertion:

- **Assert.assertEquals(str, str2, "Values are not matching.");** compares the **content** of both strings. Since the content of str and str2 is the same ("Student"), the assertion will pass.
- This method compares the actual **values** of the strings and not their memory references.

Key Differences:

1. **Memory Location:**
    - str refers to a string in the **String pool**.
    - str2 refers to a **newly created string object** in the heap memory.
2. **Equality:**
    - **Content Equality (equals method):** Both str.equals(str2) and Assert.assertEquals(str, str2) will return true because they compare the **content** of the strings.
    - **Reference Equality (== operator):** If you use str == str2, it will return false because they are two different objects in memory, even though their content is identical.

Conclusion:

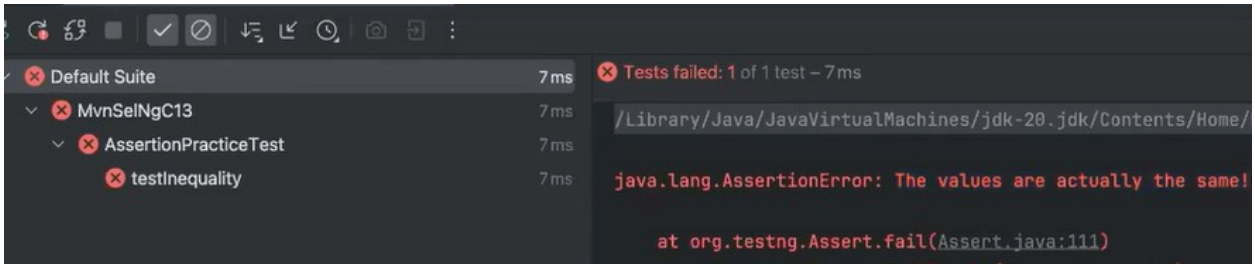
The first string (str) is stored in the String pool, while the second (str2) is stored in the heap memory.

The Assert.assertEquals method compares the content of the two strings, not their memory references, so the assertion passes.

assertNotEquals

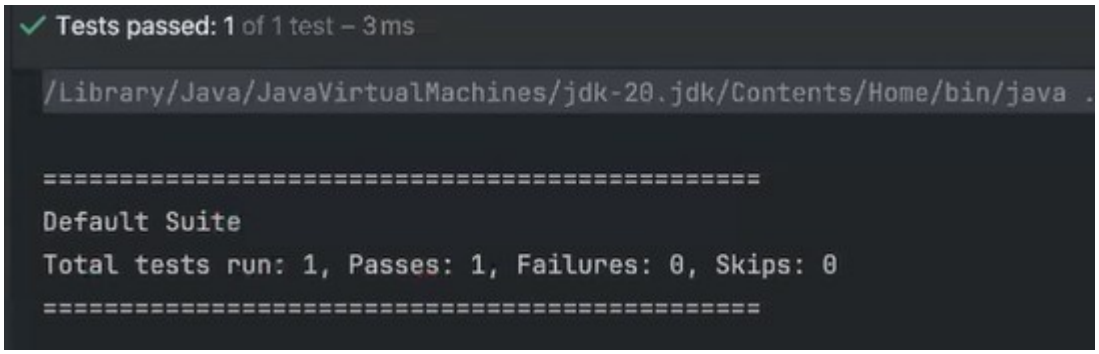
```
@Test
        public void testInequality() {
// Assert.assertNotEquals(10, 10, "The values are actually the same!");
            String str1 = "Hello!";
            String str2 = "Hello!";
            Assert.assertNotEquals(str1, str2, "The values are actually the same!");
        }
```

The test will fail because str1 and str2 are equal when using `assertNotEquals`.



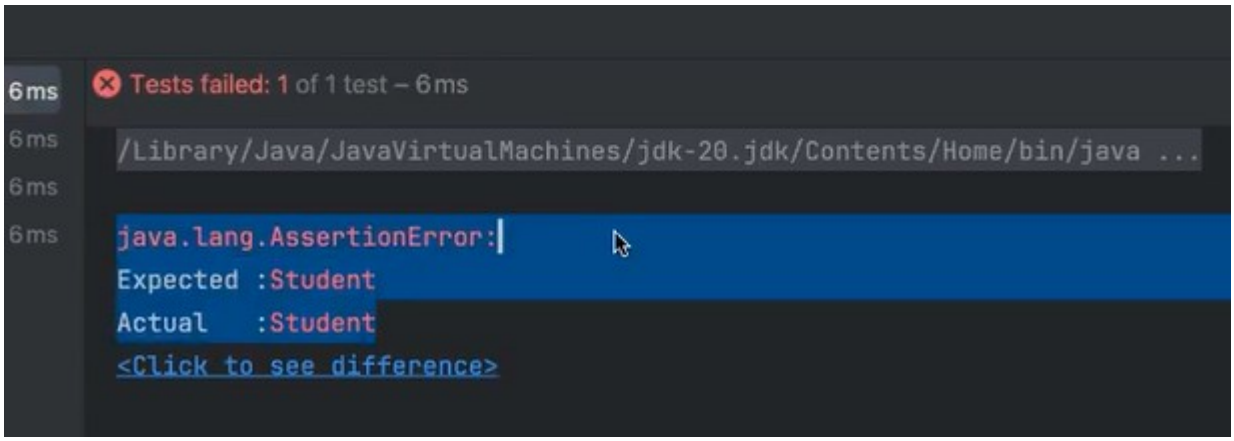

```
@Test
        public void testInequality() {
// Assert.assertNotEquals(10, 10, "The values are actually the same!");
            String str1 = "Hello !";
            String str2 = "Hello";
            Assert.assertNotEquals(str1, str2, "The values are actually the same!");
```

The test will pass because str1 and str2 are not equal when using `assertNotEquals`.


Tests passed: 1 of 1 test – 3ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0

assertSame

```
/*
      * Assert.assertSame will check if two references point to the same object
      * Are those two objects the same
      **/
@Test
        public void testIfSame() {
            String str = "Student";
            String str2 = new String("Student");
            Assert.assertSame(str, str2,"The objects are not same");
        }
```



- The code compares two strings using `Assert.assertSame(str, str2);`.
- `Assert.assertSame()` checks whether **two objects** are the **same instance** (i.e., whether they reference the same object in memory).

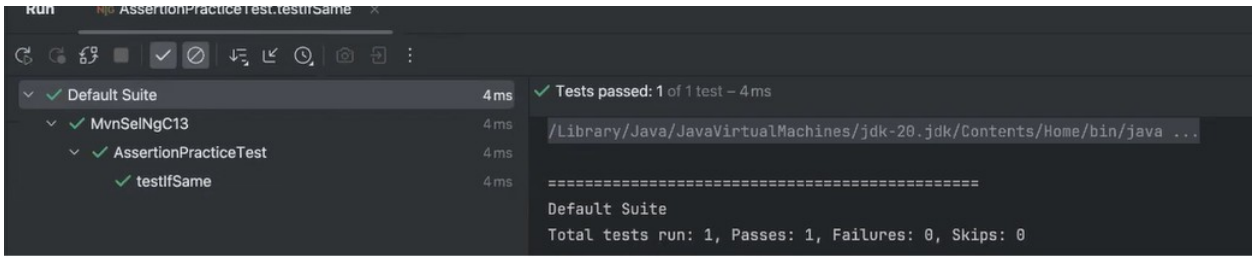
```
String str = "Student";
String str2 = new String("Student");
```

- `str` is a string literal, so it is stored in the **String pool**.
- `str2` is a new `String` object created with the `new` keyword, so it points to a **different object** in memory, even though the content is the same (`"Student"`).

Reason for the failure:

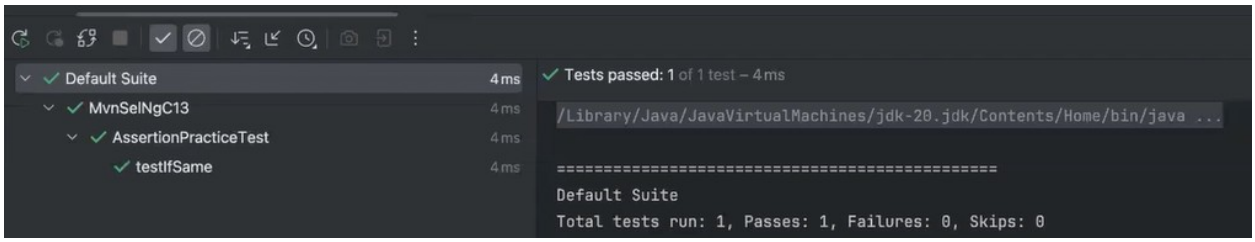
- `Assert.assertSame(str, str2)` fails because `str` and `str2` are **not the same object**. They have the same content, but they do not refer to the same memory location.
-

```
@Test
        public void testIfSame() {
            String str = "Student";
            String str2 = new String("Student");
            String str3 = "Student";
            Assert.assertSame(str, str3, "The objects are not same");
        }
```

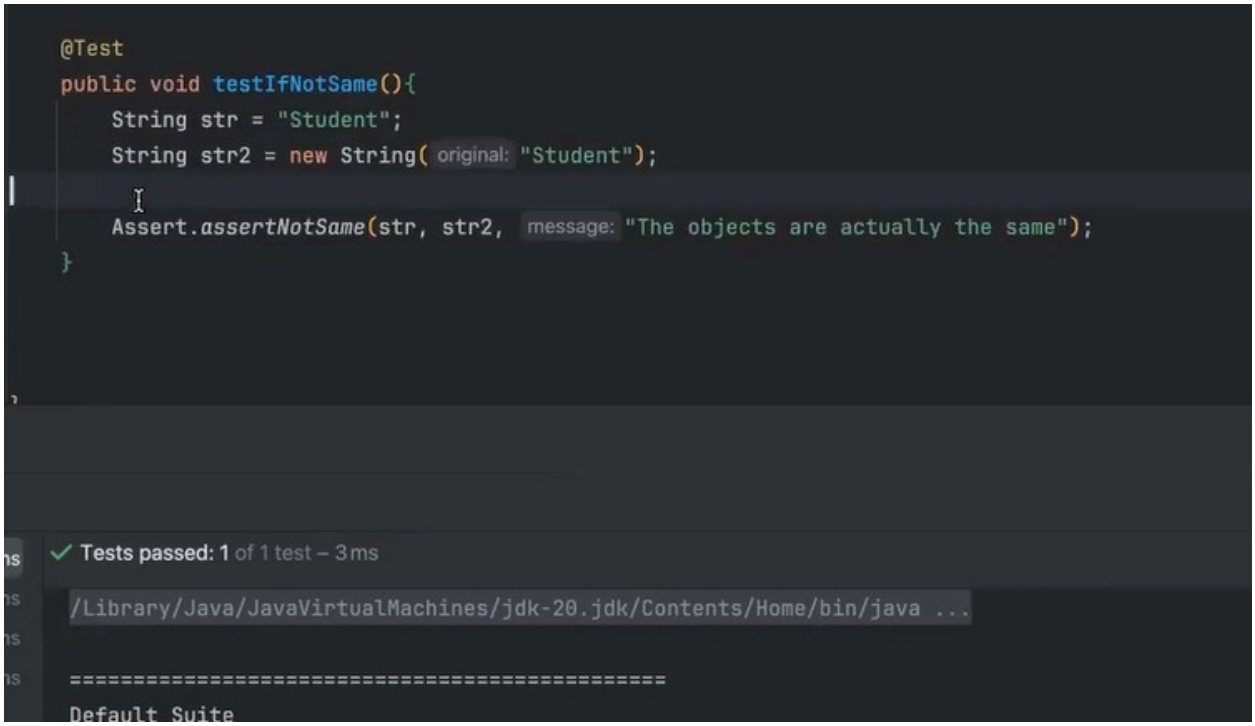


@Test

```java
public void testIfSame() {
    String str = "Student";
    String str2 = new String("Student");
    str3 = str;
    Assert.assertSame(str, str2, "The objects are not same");
}
```

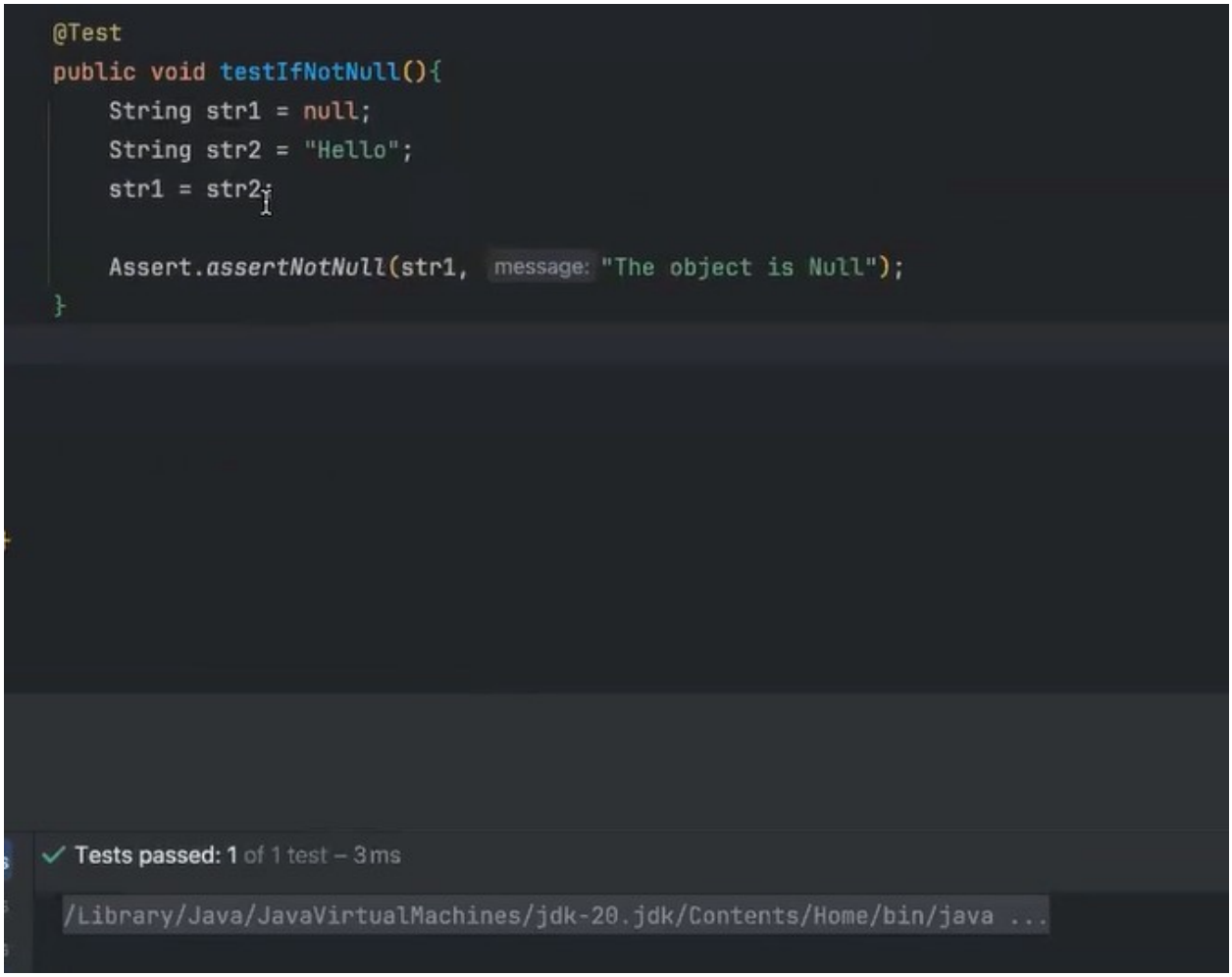


assertNotSame


@Test
public void testIfNotSame(){
String str = "Student";
String str2 = new String(original: "Student");
Assert.assertNotSame(str, str2, message: "The objects are actually the same");
}
Tests passed: 1 of 1 test – 3ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Default Suite

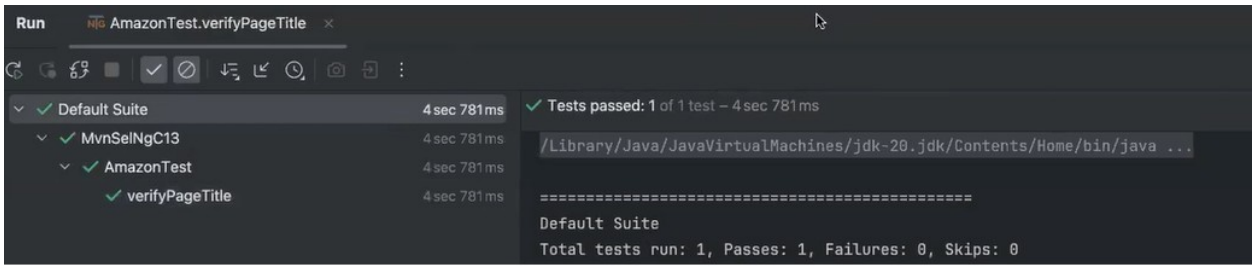
assertNull

assertNotNull

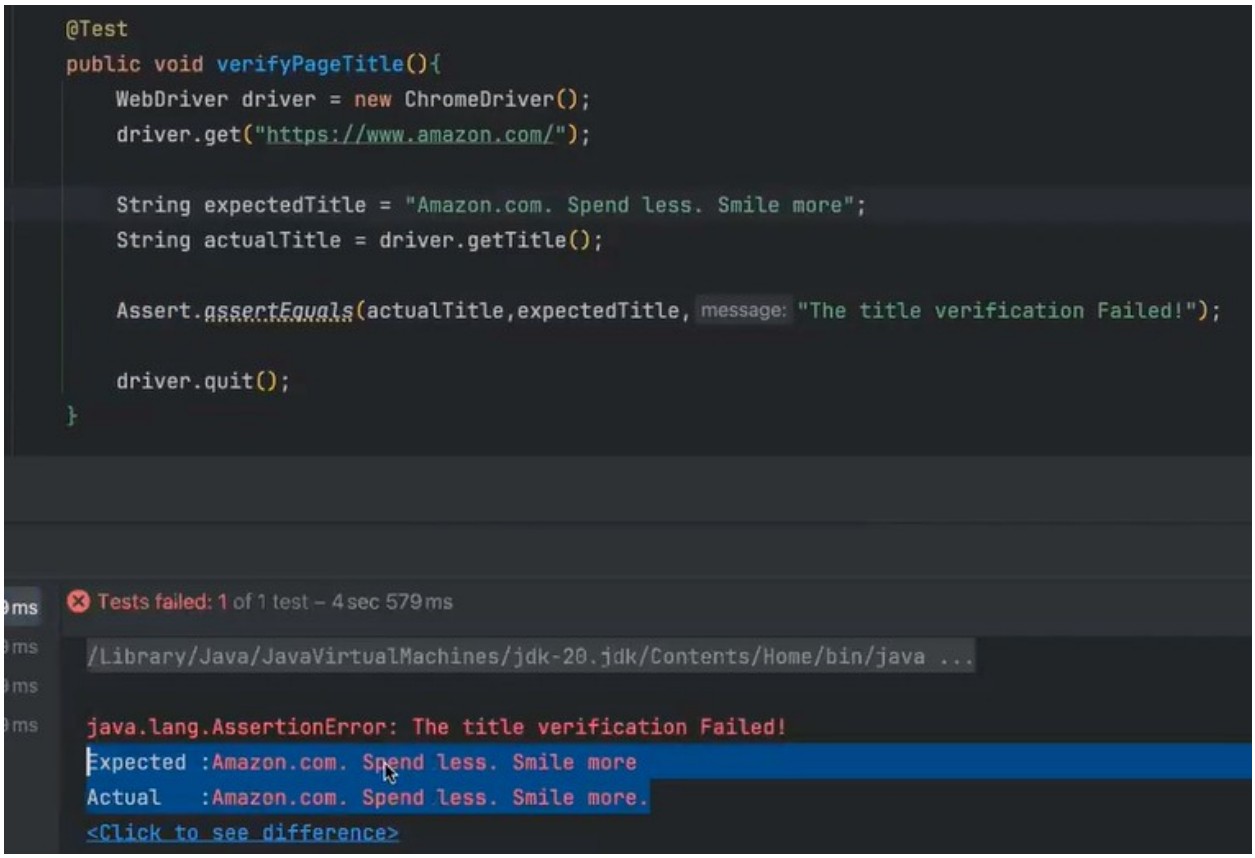


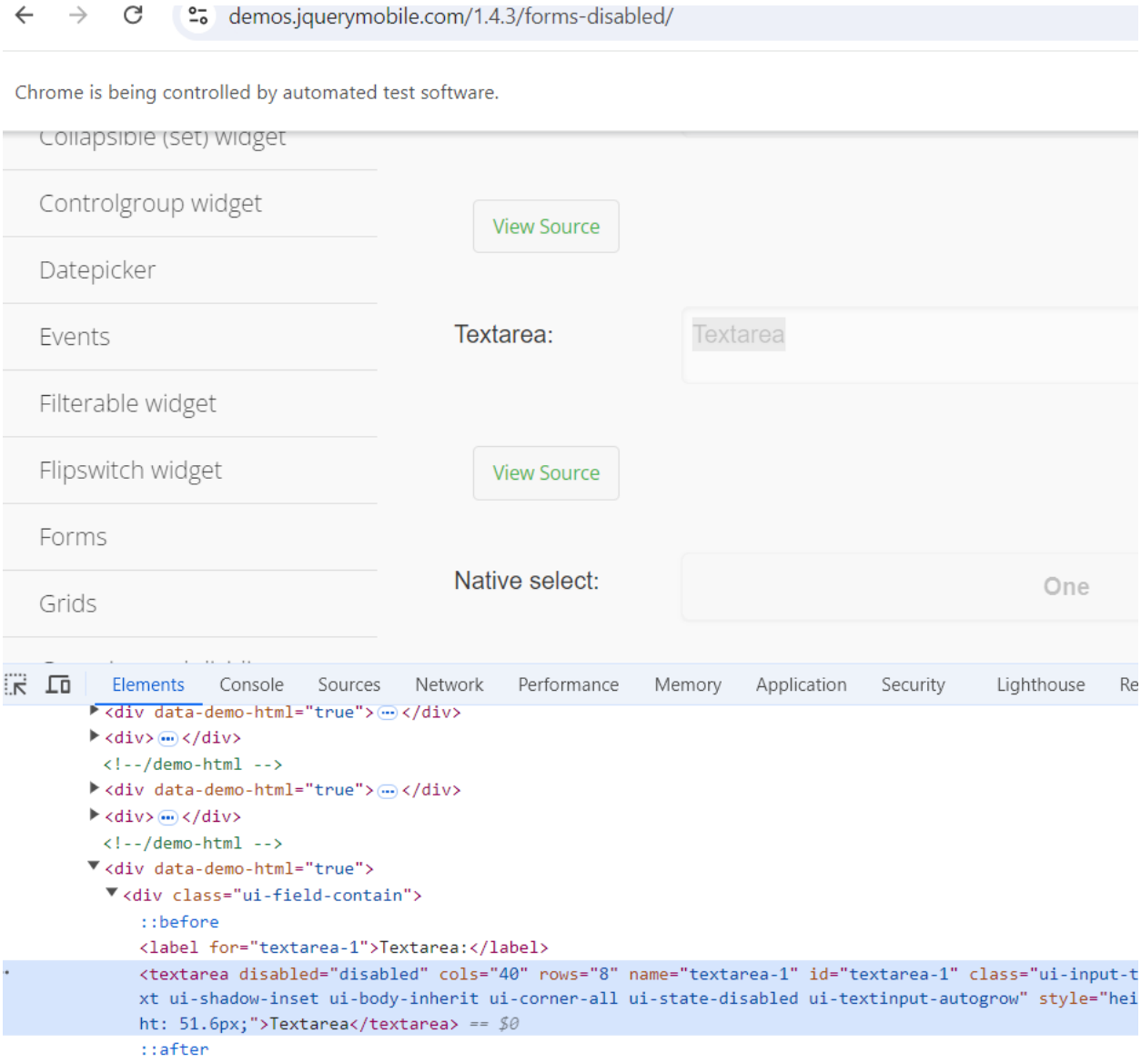

```java
package assertions;

import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class verifypagetitle {
	// Write a method
	// Navigate to Amazon homepage
	/*
	 * Verify page title with assertEquals() method Run your code with @Test
	 * annotation Given user navigates to "https://www.amzon.com" When user lands on
	 * Landing Page Then title should be "Amazon.com. Spend less. Smile more."
	 *
	 **/
	@Test
	public void verifyPageTitle() {
		WebDriver driver = new ChromeDriver();
		driver.get("https://www.amazon.com/");
		String expectedTitle = "Amazon.com. Spend less. Smile more.";
		String actualTitle = driver.getTitle();
```

```
            Assert.assertEquals(actualTitle, expectedTitle, "The title verification
Failed!");
            driver.quit();
      }
}
```


Run
AmazonTest.verifyPageTitle
Default Suite 4 sec 781 ms
MvnSelNgC13 4 sec 781 ms
AmazonTest 4 sec 781 ms
verifyPageTitle 4 sec 781 ms
Tests passed: 1 of 1 test – 4 sec 781 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0

When it fails:


@Test
public void verifyPageTitle(){
WebDriver driver = new ChromeDriver();
driver.get("https://www.amazon.com/");
String expectedTitle = "Amazon.com. Spend less. Smile more";
String actualTitle = driver.getTitle();
Assert.assertEquals(actualTitle,expectedTitle, message: "The title verification Failed!");
driver.quit();
}
Tests failed: 1 of 1 test – 4 sec 579 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
java.lang.AssertionError: The title verification Failed!
Expected :Amazon.com. Spend less. Smile more
Actual :Amazon.com. Spend less. Smile more.
<Click to see difference>




```
package assertions;

import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class JQueryTest {

	/*
	 * Go to https://demos.jquerymobile.com/1.4.3/forms-disabled/ Assert that
	 * textArea field is disabled
	 *
	 * Given user navigates to
	 * "https://demos.jquerymobile.com/1.4.3/forms-disabled/" When user lands on
```

```
	 * Form Disabled Page Then textArea field is disabled
	 **/
	@Test
	public void verifyTextAreaDisabled() {
		WebDriver driver = new ChromeDriver();
		driver.get("https://demos.jquerymobile.com/1.4.3/forms-disabled/");
		WebElement textArea = driver.findElement(By.id("textarea-1"));
		textArea.isEnabled();
// If a tag is disabled it will have attribute disabled
// In selenium we can only identify if a tag is Enabled.
// isEnabled() will give us back true if we can interact with the element
// and false if we cannot
// The test is about we verify that we cannot interact with element
		// The test is about we verify that we cannot interact with element
		Assert.assertFalse("The text area is not disabled", textArea.isEnabled());
		// The test is about we verify that we cannot interact with element
		Assert.assertFalse("The text area is not disabled", textArea.isEnabled());
		driver.quit();

		driver.quit();
	}
}
```

By using textArea.isEnabled => False

by using Assert.*assertFalse()= True*

VERIFY - SOFT ASSERT

- When using **hard Assert**, if assertion fails, then an Assertion exception is thrown and test stops executing that @Test method. .
  - It **marks that @Test as failed**, and jumps to execution of next @Test methods.

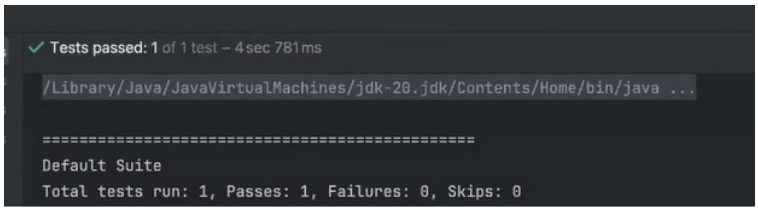
- **soft Assert**, it does NOT throw an exception when an assert fails and would continue with the next step after the assert statement.
  - If using SoftAssert, do not forget to include **assertAll()** method call at at the end. Otherwise, failed tests will show as passed.

TASK 1

```
	/*
	 * Scenario: Verify employee details against database in the Employee
	 * Information Page Given user navigates to Orange HRM Login page When user logs
	 * in with username "yoll-student" and password "#Bootcamp5" Then user should be
```

```
       * redirected to the Dashboard page When user navigates to the PIM tab Then
       * Employee Information Page should be displayed When user searches for employee
       * with ID "136034" And fetches the employee details from the database for ID
       * "136034" Then displayed employee details should match the data from the
       * database I
       */

package assertions;

import org.junit.Assert;
import org.junit.Test;

public class SoftAssertPracticeTest {

      @Test
      public void verifyEmployeeDetails() {
            /* Selenium Scripts To Fetch/Extract the data from UI for employee id 136034
*/
            String uiFirstName = "Test";
            String uiLastName = "Automan";
            String uiJobTitle = "SDET";
            String uiEmployeeStatus = "Full Time";
            String uiSubUnit = "";
            String uiSupervisor = "";
            /*
             * Connect to DB and extract the record for an employee with the same id
136034
             */


            String dbFirstName = "Test";
            String dbLastName = "Automan";
            String dbJobTitle = "SDET";
            String dbEmployeeStatus = "Full Time";
            String dbSubUnit = "";
            String dbSupervisor = "";

            /* Verify the data points for the record */
            Assert.assertEquals(uiFirstName, dbFirstName, "The first name values don't
match!");
            Assert.assertEquals(uiLastName, dbLastName, "The last name values don't
match!");
            Assert.assertEquals(uiJobTitle, dbJobTitle, "The job title values don't
match!");
            Assert.assertEquals(uiEmployeeStatus, dbEmployeeStatus, "The employee status
values don't match!");
            Assert.assertEquals(uiSubUnit, dbSubUnit, "The sub unit values don't
match!");
            Assert.assertEquals(uiSupervisor, dbSupervisor, "The supervisor values don't
match!");
      }
}
```

Result :


Tests passed: 1 of 1 test – 4 sec 781 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
==========
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0

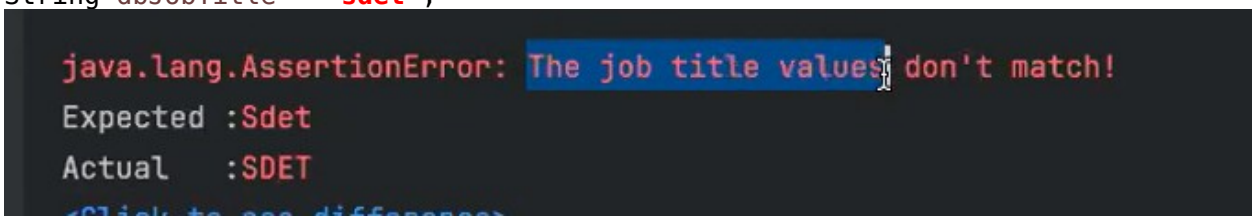
TASK 2 : Bad approach

What happens if we find a bug/bugs ?

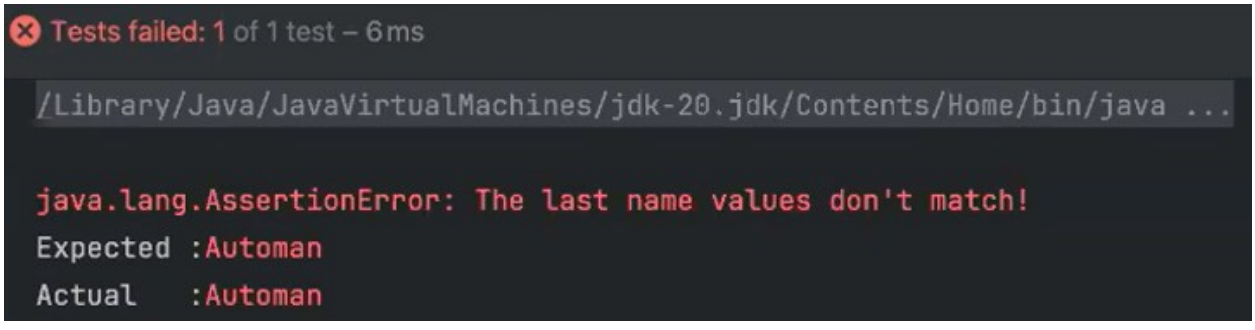
Be careful: it shows only one bug not the all bugs.

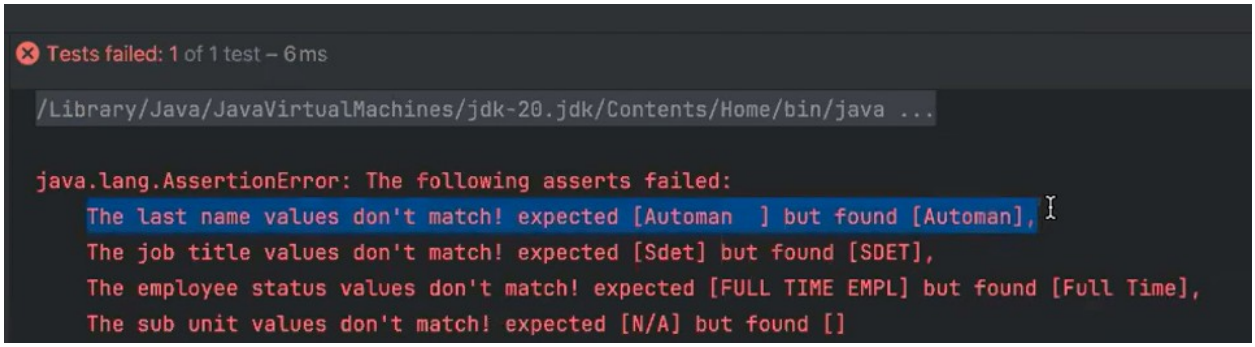
Example: if we have three bugs: Selenium scripts does not mathch DB Scripts:

```
String uiFirstName = "Test";
String uiLastName = "Automan";
String uiJobTitle = "SDET";

String dbFirstName = "test";
String dbLastName = "  Automan  ";
String dbJobTitle = "sdet";
```



When we fix a bug, we see the other bug after the previous is fixed !
When run the test, we find another bug because the tester cannot see it, and reports only one bug.


Tests failed: 1 of 1 test – 6 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
java.lang.AssertionError: The last name values don't match!
Expected :Automan
Actual :Automan


Tests failed: 1 of 1 test – 6 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
java.lang.AssertionError: The following asserts failed:
The last name values don't match! expected [Automan] but found [Automan],
The job title values don't match! expected [Sdet] but found [SDET],
The employee status values don't match! expected [FULL TIME EMPL] but found [Full Time],
The sub unit values don't match! expected [N/A] but found []

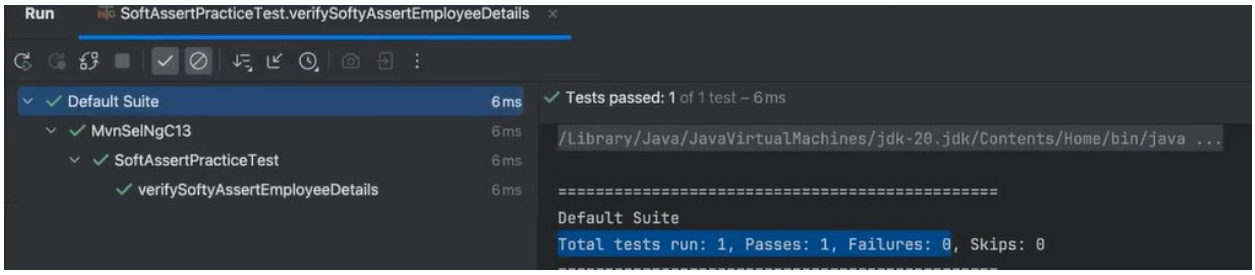
Soft Assert: Good Approach

```
package assertions;

import org.junit.Assert;
import org.junit.Test;
import org.testng.asserts.SoftAssert;

public class SoftAssertPracticeTest {

	/*
	 * Scenario: Verify employee details against database in the Employee
	 * Information Page Given user navigates to Orange HRM Login page When user logs
	 * in with username "yoll-student" and password "#Bootcamp5" Then user should be
	 * redirected to the Dashboard page When user navigates to the PIM tab Then
	 * Employee Information Page should be displayed When user searches for employee
	 * with ID "136034" And fetches the employee details from the database for ID
	 * "136034" Then displayed employee details should match the data from the
	 * database I
	 */
	@Test
	public void verifyEmployeeDetails() {
		/* Selenium Scripts To Fetch/Extract the data from UI for eplovee id 136034
*/
		String uiFirstName = "Test";
		String uiLastName = "Automan";
		String uiJobTitle = "SDET";
		String uiEmployeeStatus = "Full Time";
		String uiSubUnit = "";
		String uiSupervisor = "";
		/*
		 * Connect to DB and extract the record for an employee with the same id
136034
		 */
		String dbFirstName = "tEST";
		String dbLastName = "  Automan  ";
		String dbJobTitle = "sdet";
		String dbEmployeeStatus = "FullTime";
		String dbSubUnit = "";
		String dbSupervisor = "";
```

```
            // Create an object of SoftAssert
            SoftAssert softAssert = new SoftAssert();
            // Further the softAssert object will have exactly the same methods as Assert
            // class
            // assertTrue False
            // assertEquals | NotEquals
            /* Verify the data points for the record */
            softAssert.assertEquals(uiFirstName, dbFirstName, "The first name values
don't match!");
            softAssert.assertEquals(uiLastName, dbLastName, "The last name values don't
match!"); // at this line the code
            softAssert.assertEquals(uiJobTitle, dbJobTitle, "The job title values don't
match!");
            softAssert.assertEquals(uiEmployeeStatus, dbEmployeeStatus, "The employee
status values don't match!");
            softAssert.assertEquals(uiSubUnit, dbSubUnit, "The sub unit values don't
match!");
            softAssert.assertEquals(uiSupervisor, dbSupervisor, "The supervisor values
don't match!");

            // Lastly, if you want to have the results of all assertion use: assertAll()
            // To evaluate the assertion use that assertAll() stmt
            softAssert.assertAll();
      }
}
```

If we do not put : `assertAll()`

The test will pass ! because we did not validate it, and we do want that !

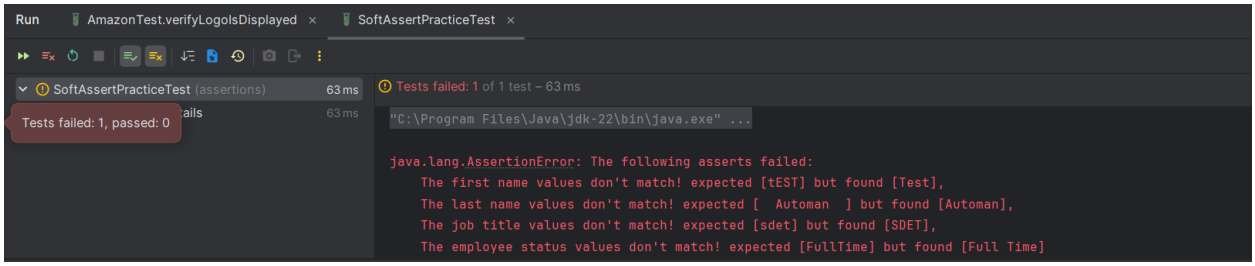


We must put : assertAll to evaluate the assertion

By this way, we can know the whole bugs that must be fixed by the developpers, and can execute the whole test and move forward even if a test fails without stopping, and can avoid fixing one by one as the previous test : *TASK 2*

In Task 2, if the test fails, it will not be able to move forward.

The expected result:


Run
AmazonTest.verifyLogoIsDisplayed
SoftAssertPracticeTest
SoftAssertPracticeTest (assertions) 63 ms
Tests failed: 1, passed: 0
Tests failed: 1 of 1 test – 63 ms
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
java.lang.AssertionError: The following asserts failed:
The first name values don't match! expected [tEST] but found [Test],
The last name values don't match! expected [Automan] but found [Automan],
The job title values don't match! expected [sdet] but found [SDET],
The employee status values don't match! expected [FullTime] but found [Full Time]

```
UI Automation:

From the perspective of writing goo automation scripts we need to have independant test cases
Whe the test cases are independent they are usally consize
Means we should have 1-2 verifications per test
- Verify is a button is present
- Verify text area is disabled

In UI we will use mostly Hard Assert:
- assertTrue | False
- assertEquals | NotEquals

In companies usually there is Code of Conduct.
If they set that using soft assert is somthing they ussualy utilize

Soft assertions are used specifically morew when there are multiple verifications required:
- When there is a multiple data check
- E2E cases - Whole flow of user

This applies specifically to the framework of TestNG with Selenium
```

QUESTION AND ANSWERS:

What is the whole purpose of TestNG?

TestNG is testing framework that initially was developed for Unit Test.

Its main job is to help organize and manage software tests, especially for big projects.

It lets you run lots of different tests easily, find out which ones passed or failed.

Why we need TestNG as a Test Automation Engineer?

TestNG is useful for automation testers because it makes testing faster, more organized, and reliable.

It has features like grouping tests, deciding the order of tests, and repeating tests, which help in checking software thoroughly and efficiently.

What is an assertion and why we need it?

An assertion is like a check in a test to see if something is true.

We need assertions to make sure the software works as expected.

If the assertion fails (finds a problem), it tells us there's something wrong that needs fixing.

What Assertion Types TestNG provides?

TestNG provides two main types of assertions - Hard Assert and Soft Assert.

What Assertion methods are there?

Some of the most used assertion methods would be:

- assertTrue()
- assertFalse()
- assertEquals()
- assertNotEquals()

Few more assertions that we know:

- assertNull()
- assertNotNull()
- assertSame()
- assertNotSame()
- fail()

▼ What are the differences between Hard Assert and Soft Assert?

Hard Assert

1. Hard assert fails the test case immediately and doesn't run the subsequent code.
   If we have more assertions after the the failed one, the subsequent assertions will not be evaluated.

2. Hard assert contains static methods, so to use them we call them by class name:

```
Assert.assertTrue(condition, message);
```

3. To get the results of Hard assert we don't need anything additionally. We simply call the assert methods and they will give us the result if test pass of fail.

Soft Assert

1. Soft assert will not fail immediately when an assertion fails, instead it collects the information about assertions and will give an result when we call assertAll() method.
   If we have more assertions after one failed they will be evaluated and given the result based on all of the assertions.

2. Soft Assert contains instance methods. In order to use them we need to create and object of SoftAssert and then use object name to call the methods:

```
SoftAssert softAssert = new SoftAssert();
softAssert.assertTrue(actual, message);
```

3. With soft assertions we will need to use assertAll() method after we have completed all the verifications in order to get a result if test passes or fails

▼ **What are the use cases for Hard and Soft Assertions?**

Hard assertions are used in most of the cases. They allow you to report immediately an issue.

Soft Asserts are useful when the test case involves verifying multiple data points or creating flows.

1. How can we improve our code/automation with testNG features

2. How do we disable tests that are not ready to be executed

3. How do we prioritize test in TestNG

Differences Between the Smoke and Regression Tests:

1. **Smoke Test**: Focuses on basic functionality and stability (e.g., homepage loads, login works). It's designed to catch major issues before deep testing.
2. **Regression Test**: Covers a broader scope and ensures that both new and existing functionalities work properly after changes are made to the application.

The **smoke tests** are simpler and faster, while **regression tests** are more comprehensive and can be time-consuming. Both are essential for ensuring software quality.

Here's an example of both a **smoke test** and a **regression test** using Selenium and TestNG.

Smoke Test Example:

Let's assume we are testing an e-commerce site. The smoke test will focus on critical functionalities like whether the homepage loads correctly and whether the user can log in.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class SmokeTest {
    WebDriver driver;

    @BeforeMethod
    public void setup() {
        // Set up ChromeDriver
        System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }

    @Test
    public void testHomePageLoad() {
        // Navigate to e-commerce homepage
        driver.get("https://www.your-ecommerce-site.com");
        // Verify that the homepage title is correct
        String expectedTitle = "Your E-Commerce Site";
        String actualTitle = driver.getTitle();
        Assert.assertEquals(actualTitle, expectedTitle, "Homepage did not load
correctly.");
    }

    @Test
    public void testUserLogin() {
        // Navigate to the login page
        driver.get("https://www.your-ecommerce-site.com/login");

        // Find username and password input fields and login button
        WebElement usernameField = driver.findElement(By.id("username"));
        WebElement passwordField = driver.findElement(By.id("password"));
        WebElement loginButton = driver.findElement(By.id("login-btn"));

        // Enter credentials and click login
        usernameField.sendKeys("testuser");
```

```
        passwordField.sendKeys("password123");
        loginButton.click();

        // Verify successful login by checking the presence of a logout button
or dashboard
        WebElement logoutButton = driver.findElement(By.id("logout-btn"));
        Assert.assertTrue(logoutButton.isDisplayed(), "Login failed.");
    }

    @AfterMethod
    public void teardown() {
        if (driver != null) {
            driver.quit();
        }
    }
}
```

Explanation of Smoke Test:

- **Test 1:** `testHomePageLoad()`: Verifies that the homepage loads by checking the page title. This is a very basic check to ensure the site is up and running.
- **Test 2:** `testUserLogin()`: Verifies that a user can log in successfully. This checks one of the most critical user flows.
- These tests check fundamental aspects of the application, ensuring that the build is stable for further testing.

Regression Test Example:

Let's assume that after adding a new feature (e.g., a new "Add to Cart" functionality), you want to ensure that both the new feature and older features still work properly. The regression test will cover more scenarios and deeper testing.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class RegressionTest {
```

```
    WebDriver driver;

    @BeforeMethod
    public void setup() {
        // Set up ChromeDriver
        System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }

    @Test
    public void testProductSearch() {
        // Navigate to e-commerce site
        driver.get("https://www.your-ecommerce-site.com");

        // Search for a product
        WebElement searchBox = driver.findElement(By.id("search-box"));
        searchBox.sendKeys("laptop");
        searchBox.submit();

        // Verify that the search results contain the product
        WebElement searchResult = driver.findElement(By.cssSelector(".product-
title"));
        Assert.assertTrue(searchResult.getText().contains("laptop"), "Product
search failed.");
    }

    @Test
    public void testAddToCartFunctionality() {
        // Navigate to a product page
        driver.get("https://www.your-ecommerce-site.com/product/1234");

        // Click on 'Add to Cart' button
        WebElement addToCartButton = driver.findElement(By.id("add-to-cart-
btn"));
        addToCartButton.click();

        // Verify that the item has been added to the cart
        WebElement cartCount = driver.findElement(By.id("cart-count"));
        Assert.assertEquals(cartCount.getText(), "1", "Item was not added to the
cart.");
    }

    @Test
    public void testCheckoutProcess() {
        // Navigate to checkout page
        driver.get("https://www.your-ecommerce-site.com/checkout");

        // Verify that checkout page loads correctly
        WebElement checkoutButton = driver.findElement(By.id("checkout-btn"));
```

```
        Assert.assertTrue(checkoutButton.isDisplayed(), "Checkout process
failed.");
    }

    @AfterMethod
    public void teardown() {
        if (driver != null) {
            driver.quit();
        }
    }
}
```

Explanation of Regression Test:

- **Test 1:** `testProductSearch()`: Ensures that the product search functionality is working as expected, which is a core feature of the application.
- **Test 2:** `testAddToCartFunctionality()`: Verifies that the newly added "Add to Cart" functionality works as expected.
- **Test 3:** `testCheckoutProcess()`: Checks whether the checkout process works as expected.
- These tests ensure that both the existing features (like search and checkout) and any newly added functionality work properly after changes to the codebase.

Positive and Negative Test Cases Approach

Positive and negative test cases are used in software testing to ensure that an application functions correctly under both normal and abnormal conditions. Let's break down what each approach involves:

---

1. Positive Test Cases

Positive testing is intended to validate that the application behaves as expected when provided with valid and correct inputs or scenarios. It focuses on testing the application with expected, normal inputs to verify that the system does what it's supposed to do.

Characteristics of Positive Test Cases:

- Tests for correct behavior under normal conditions.
- Uses valid input data.
- Ensures the system meets the specified requirements and works as intended.

Example Scenarios of Positive Test Cases:

- **Login Feature**: Enter valid username and password; verify that the user successfully logs in.
- **Form Submission**: Fill in all required fields with valid data; check that the form is submitted successfully.
- **Search Functionality**: Enter a valid search term and verify that relevant results are returned.

Example: Positive Test Case for a Login Functionality:

```
@Test
public void testValidLogin() {
    WebDriver driver = new ChromeDriver();
    driver.get("https://example.com/login");

    // Enter valid credentials
    WebElement username = driver.findElement(By.id("username"));
    WebElement password = driver.findElement(By.id("password"));
    username.sendKeys("validUser");
    password.sendKeys("validPassword");

    // Submit login form
    driver.findElement(By.id("login-button")).click();

    // Assert successful login by checking the presence of a logout button
    WebElement logoutButton = driver.findElement(By.id("logout-button"));
    Assert.assertTrue(logoutButton.isDisplayed(), "Login failed.");

    driver.quit();
}
```

---

2. Negative Test Cases

Negative testing is intended to validate that the application gracefully handles invalid or unexpected inputs and conditions. The goal is to ensure that the system doesn't crash and responds properly when encountering invalid data, and that it correctly enforces constraints and validations.

Characteristics of Negative Test Cases:

- Tests the system’s behavior under invalid, incorrect, or unexpected conditions.
- Uses invalid inputs to provoke errors or invalid responses.
- Ensures that the system handles errors gracefully, without crashing or producing unpredictable results.

Example Scenarios of Negative Test Cases:

- **Login Feature**: Enter an invalid username and password, then verify that an error message is displayed.
- **Form Submission**: Leave mandatory fields empty and check whether an appropriate error message is displayed.
- **File Upload**: Try to upload a file that exceeds the size limit or has an unsupported format; verify that the system rejects the upload with an appropriate message.

Example: Negative Test Case for a Login Functionality:

```
@Test
public void testInvalidLogin() {
    WebDriver driver = new ChromeDriver();
    driver.get("https://example.com/login");

    // Enter invalid credentials
    WebElement username = driver.findElement(By.id("username"));
    WebElement password = driver.findElement(By.id("password"));
    username.sendKeys("invalidUser");
    password.sendKeys("invalidPassword");

    // Submit login form
    driver.findElement(By.id("login-button")).click();

    // Assert an error message is displayed for invalid login
    WebElement errorMessage = driver.findElement(By.id("error-message"));
    Assert.assertTrue(errorMessage.isDisplayed(), "Error message was not displayed for
invalid login.");

    driver.quit();
}
```

Key Differences Between Positive and Negative Testing:

Aspect	Positive Testing	Negative Testing
Focus	Tests valid inputs and expected behavior	Tests invalid inputs and unexpected conditions
Objective	Verifies the system behaves as expected in normal scenarios	Verifies the system handles errors gracefully
Outcome	Confirms the correct functionality of the application	Confirms that the system doesn't crash or misbehave
Example	Logging in with valid credentials	Logging in with invalid credentials
Typical Result	The system successfully performs the expected action	The system should display an error or reject the action

When to Use Positive and Negative Testing:

- **Positive Testing**: Is used in day-to-day testing cycles to validate that features and functionalities behave as expected.
- **Negative Testing**: Is critical during regression testing, system hardening, or security testing phases where you need to ensure the system handles erroneous or malicious inputs correctly.

By adopting both approaches, you'll ensure that your application not only meets requirements under normal conditions but also gracefully handles unexpected input, enhancing its reliability and robustness.

```java
package practice;


import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.Test;

    public class HRMLoginTestTwo {


        /*
    User should be able to successfully login
    */
    @Test
    public void testSuccessfulLoginDisplaysUsername() throws InterruptedException {
        /*
        Scenario: User should be able to successfully login
        • Given User Navigates to login page
        · When User Enters valid username: yoll-student
        • And User Enters valid password: Bootcamp5#
        • Then account username is displayed as "Welcome Yoll"
        */

        // Set Up
        WebDriver driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");

        // Locate the elements
        WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
        WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
        WebElement loginButton = driver.findElement(By.id("btnLogin"));

        // Perform login
        usernameInputBox.sendKeys("yoll-student");
        passwordInputBox.sendKeys("Bootcamp5#");
        loginButton.click();

        // Wait for the welcome message to appear
        Thread.sleep(2000);
```

```
        // Validate the welcome message
        WebElement usernameMessage = driver.findElement(By.id("welcome"));
        String expectedUsername = "Welcome Yoll";
        String actualUsername = usernameMessage.getText();

        Assert.assertEquals("Username verification failed.", expectedUsername,
actualUsername);

        // Tear down
        driver.quit();
    }

    /*
    User should not be able to login to HRM with invalid credentials
    */
    @Test
    public void testLoginWithInvalidCredentialsDisplaysErrorMessage() throws
InterruptedException {
        /*
        Scenario: User should not be able to login to HRM with invalid credentials
        Given User navigates to login page
        When User enters "invalid" username
        And User enters "invalid" password
        And User clicks on login button
        Then User receives an error message as "Invalid credentials"
        */

        // Set Up
        WebDriver driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");

        // Locate the elements
        WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
        WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
        WebElement loginButton = driver.findElement(By.id("btnLogin"));

        // Perform login with invalid credentials
        usernameInputBox.sendKeys("invalid");
        passwordInputBox.sendKeys("invalid");
        loginButton.click();

        // Wait for the error message to appear
        Thread.sleep(2000);

        // Validate the error message
        WebElement errorMessage = driver.findElement(By.id("spanMessage"));
        String expectedErrorMessage = "Invalid credentials";
        String actualErrorMessage = errorMessage.getText();

        Assert.assertEquals("Error message verification failed.", expectedErrorMessage,
actualErrorMessage);

        // Tear down
```

```
        driver.quit();
    }
}
```

Case :

If expexted error message is not as the actual one:

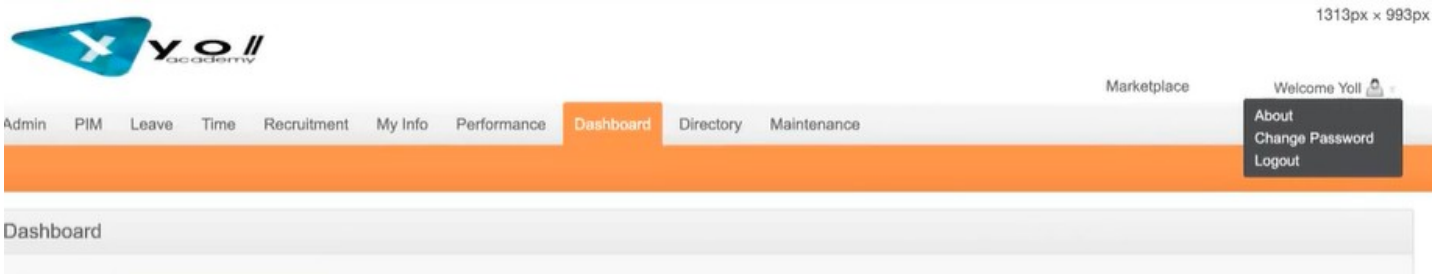
```
        WebElement errorMessage = driver.findElement(By.id("spanMessage"));

        String expectedErrorMessage = "Invalid Credentials";
        String actualErrorMessage = errorMessage.getText();

        Assert.assertEquals(actualErrorMessage, expectedErrorMessage, message: "Error message verification failed.");

        driver.quit();
```

redentialsDisplaysErr...

Tests failed: 1 of 1 test – 9 sec 177 ms

```
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...

java.lang.AssertionError: Error message verification failed.
Expected :Invalid Credentials
Actual   :Invalid credentials
<Click to see difference>
```

```
/*
        Scenario: User should be able to logout from HRM
        Given User logged in and landed on the homepage
        When User clicks on the logout button
        Then User lands on the login page

        */
```



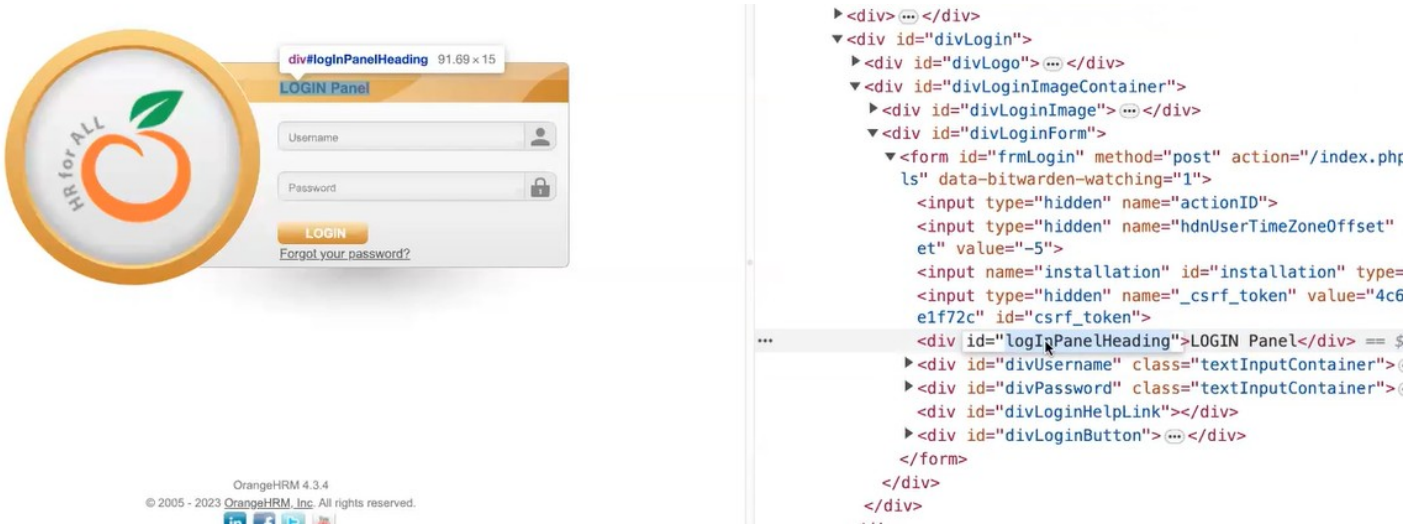

```
    @Test
        public void testLogoutRedirectsToLoginPage() throws InterruptedException {


                // Set Up
                WebDriver driver = new ChromeDriver();
                driver.manage().window().maximize();
                driver.get("http://dev-hrm.yoll.io/index.php/auth/login");

                // Locate the elements and log in
```

```
                WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
                WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
                WebElement loginButton = driver.findElement(By.id("btnLogin"));

                usernameInputBox.sendKeys("yoll-student");
                passwordInputBox.sendKeys("Bootcamp5#");
                loginButton.click();

                // Wait for the login to complete
                Thread.sleep(2000);

                // Locate and click on the dropdown menu
                WebElement dropdownMenu = driver.findElement(By.id("welcome"));
                dropdownMenu.click();
                // Wait for the login to complete
                Thread.sleep(5000);

                // Locate and click the Logout option
                WebElement logoutOption =
driver.findElement(By.xpath("//a[text()='Logout']"));
                logoutOption.click();

                // Wait for the logout to complete
                Thread.sleep(2000);

                // Validate if the login page is displayed
                WebElement loginPanel = driver.findElement(By.id("logInPanelHeading"));
                Assert.assertTrue("Logout failed or login page was not displayed.",
                        loginPanel.isDisplayed());

                // Tear down
                driver.quit();
            }
```

We need to logout, but the logout page and login page have the same name. Therefore, we cannot use this name !

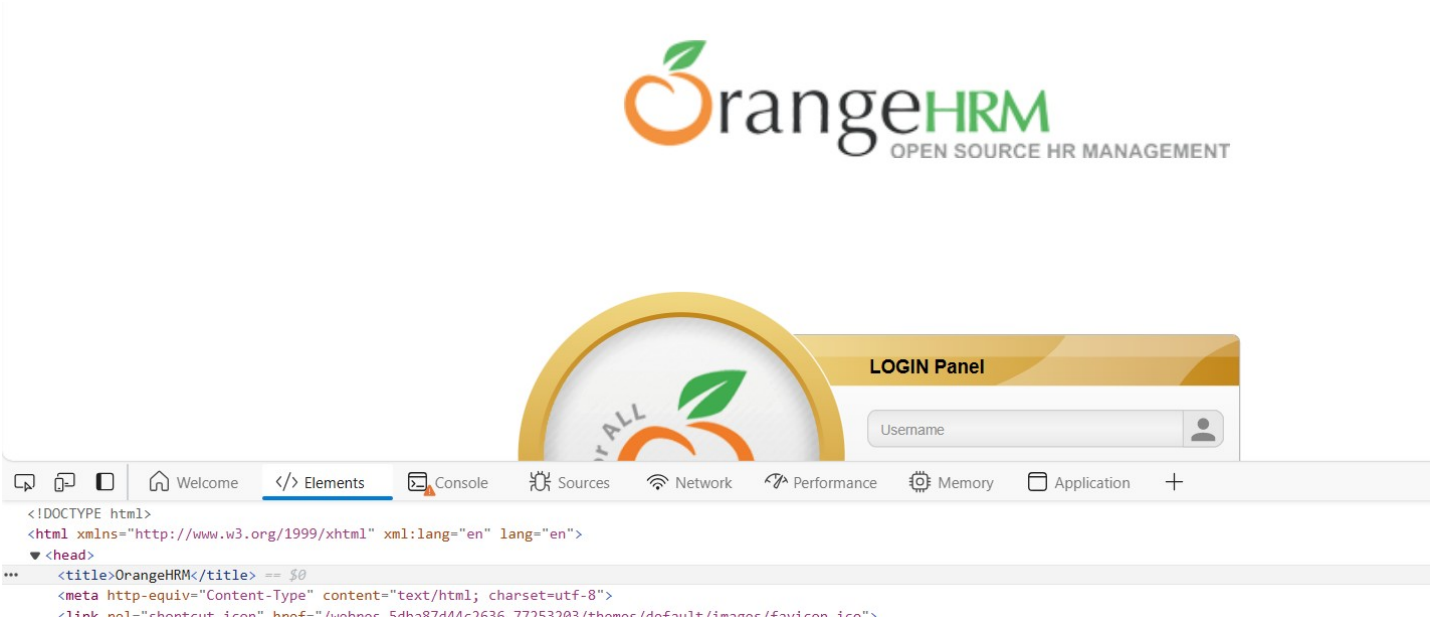
We need another approach TO MAKE SURE WE ARE THE APPROPRIATE PAGE, JUST ATTACH TO SOMETHING IS UNIQUE:

Like using Loging Panel



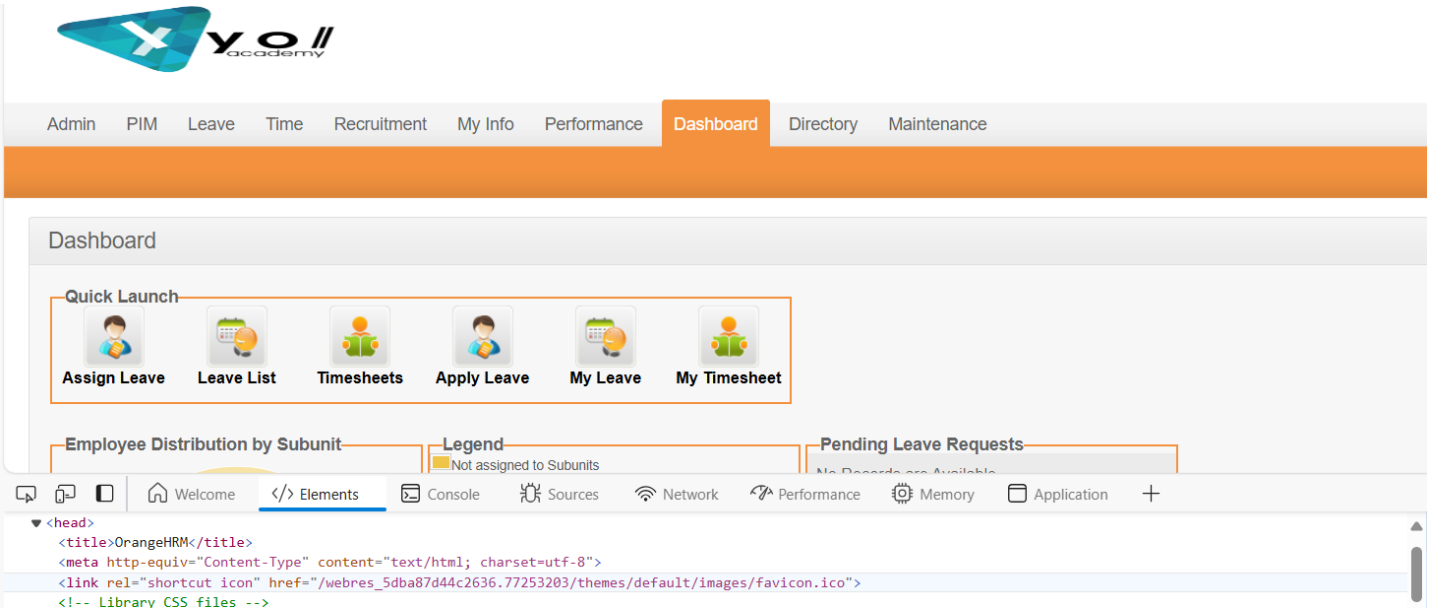
Before Login Page:

```
<title>OrangeHRM</title>
```



After login:

```
<title>OrangeHRM</title>
```



TestNG Annotations

- **@Test** → is the actual test method where we place our code to test the applications
- **@BeforeMethod** → method will be executed before executing each @Test method.
  - It executes **for each and every @Test**
- **@AfterMethod** → method will be executed after each @Test method completes its execution.
  - It executes **for each and every @Test**
- **@BeforeClass** → The annotated method will be run only once before the first test method in the current class.
- **@AfterClass** → The annotated method will be run only once after all the test methods in the current class finished running.

The same Tasks by using the annotations:

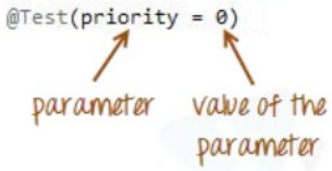
```
package practice;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.*;

public class HRMLoginTestt {
```

```
	private WebDriver driver;

	@BeforeClass
	public void dbSetUpConnection() {
		System.out.println("Database connection opened.");
	}

	/*
	 * Summary JUnit: @BeforeClass and @AfterClass must be static
	 * TestNG: @BeforeClass and @AfterClass do not need to be static. BE
CAREFUL
	 * WHEN CHOOSING THE ANNOTATIONS, IT MUST BE FOR TestNG NOT JUnit !
	 */
	@AfterClass
	public void dbCleanUp() {
		System.out.println("Database connection closed.");
	}

	@BeforeMethod
	public void setUpBrowserNavigate() {
		driver = new ChromeDriver();
		driver.manage().window().maximize();
		driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
	}

	@AfterMethod
	public void tearDown() {
		driver.quit();
	}

	@Test
	public void testSuccessfulLoginDisplaysUsername() throws InterruptedException {
		login("yoll-student", "Bootcamp5#");

		// Validate the welcome message
		WebElement usernameMessage = driver.findElement(By.id("welcome"));
		String expectedUsername = "Welcome Yoll";
		String actualUsername = usernameMessage.getText();
		Assert.assertEquals(expectedUsername, actualUsername, "Username verification
failed.");
	}

	@Test
	public void testLoginWithInvalidCredentialsDisplaysErrorMessage() throws
InterruptedException {
		login("invalid", "invalid");

		WebElement errorMessage = driver.findElement(By.id("spanMessage"));
		String expectedErrorMessage = "Invalid credentials";
		String actualErrorMessage = errorMessage.getText();
		Assert.assertEquals(expectedErrorMessage, actualErrorMessage, "Error message
verification failed.");
	}
```

```java
	@Test
	public void testLogoutRedirectsToLoginPage() throws InterruptedException {
		login("yoll-student", "Bootcamp5#");

		// Locate and click on the dropdown menu
		WebElement dropdownMenu = driver.findElement(By.id("welcome"));
		dropdownMenu.click();

		// Wait for the logout option to be available and click it
		Thread.sleep(5000);
		WebElement logoutOption =
driver.findElement(By.xpath("//a[text()='Logout']"));
		logoutOption.click();

		// Wait for the logout to complete
		Thread.sleep(2000);

		// Validate if the login page is displayed
		WebElement loginHeaderPanel = driver.findElement(By.id("logInPanelHeading"));
		Assert.assertTrue(loginHeaderPanel.isDisplayed(), "Login panel verification
failed.");
	}

	public void login(String username, String password) throws InterruptedException {
		WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
		WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
		WebElement loginButton = driver.findElement(By.id("btnLogin"));

		usernameInputBox.sendKeys(username);
		Thread.sleep(2000);
		passwordInputBox.sendKeys(password);
		Thread.sleep(2000);
		loginButton.click();
		Thread.sleep(2000);
	}
}
```

Priority & Disabling

- Java runs code sequentially unless we tell it otherwise (by creating threads/or algorithms)
- TestNG runs tests in alphabetical order.
- However, we can change that to our desired order by using priority parameter:



- We can skip any desired test case, by using enabled parameter with false value:
  - `@Test(enabled=false)`

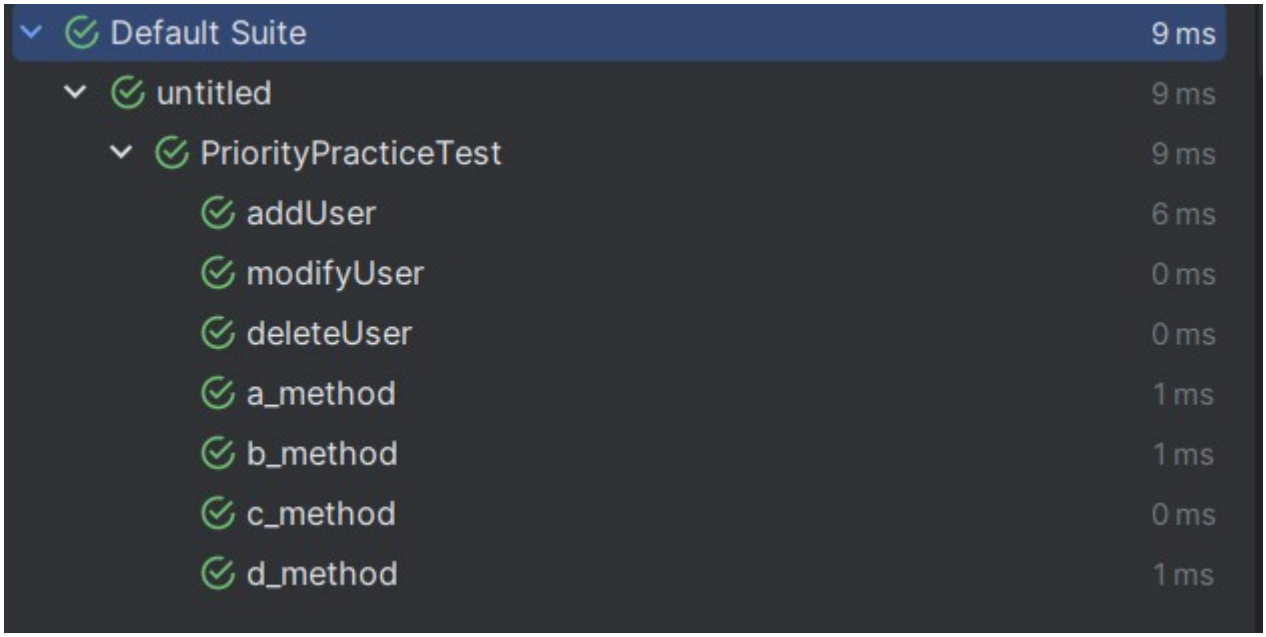
- We can use both parameters at the same time:
  - `@Test(priority=2, enabled=false)`

```java
package testNGBasics;

import org.testng.annotations.Test;

public class PriorityPracticeTest {
    /*
     * In some cases we want to control the order of the execution
     * This can be configured with Test() property priority
     * Priority starts from ZERO and it uses integers to represent it
     * By default all the Tests are set to priority @
     */
    @Test(priority = 0)
    public void addUser() {
        System.out.println("- A user has been added to the list.");
    }

    @Test(priority = 1)
    public void modifyUser() {
        System.out.println("- The user has been successfully modified.");
    }

    @Test(priority = 2)
    public void deleteUser() {
        System.out.println("- The user has been deleted.");
    }


    @Test (priority = 6)
    public void d_method() {
        System.out.println("--- method name d executed");
```

```
    }

    @Test (priority = 3)
    public void a_method() {
        System.out.println("--- method name a executed");

    }

    @Test (priority = 5)
    public void c_method() {
        System.out.println("--- method name c executed");

    }

    @Test(priority = 4)
    public void b_method() {
        System.out.println("--- method name b executed");

    }
}
```

Result:

```
- A user has been added to the list.
- The user has been successfully modified.
- The user has been deleted.
--- method name a executed
--- method name b executed
--- method name c executed
--- method name d executed

===============================================
Default Suite
Total tests run: 7, Passes: 7, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```



We can put a negative number before (priority = 0) instead of changing the Priority numbers:

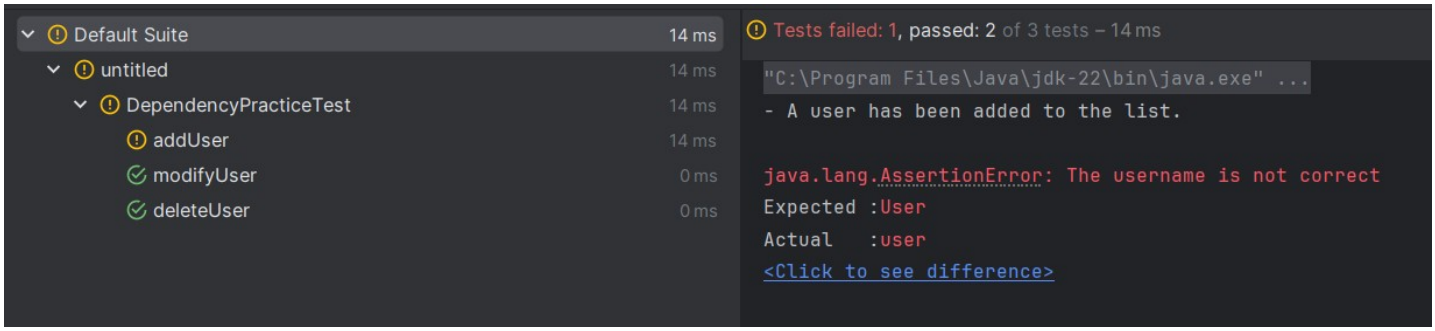
```
    @Test (priority = -1)
    public void a_method() {
        System.out.println("--- method name a executed");
```

Test Dependecy

```
/*
In some situations when you have some test cases that relate on each other
*You don't always need to run the following test cases when the initial
fails
*For that case TestNG provides you another property dependsOnMethods =
"methodName"
When you have dependency on another method
*It will not run the test case if the method we depend on fails
**/


package testNGBasics;

import org.testng.Assert;
import org.testng.annotations.Test;

public class DependencyPracticeTest {
      @Test(priority = 0)
      public void addUser() {
            System.out.println("- A user has been added to the list.");
```

```
            String expected = "User";
            String actual = "user";
            Assert.assertEquals(actual, expected, "The username is not correct");
      }

      @Test(priority = 1)
      public void modifyUser() {
            System.out.println("- The user has been successfully modified.");
      }

      @Test(priority = 2)
      public void deleteUser() {
            System.out.println("- The user has been deleted.");
      }
}
```

In this Case, addUser Test was failed, but it supposed to stop on this step, but we see another test were executed:
addUser
modifyUser
deleteUser
it does not make sense to modify and delete a user that was added !


Default Suite 14 ms
untitled 14 ms
DependencyPracticeTest 14 ms
addUser 14 ms
modifyUser 0 ms
deleteUser 0 ms
Tests failed: 1, passed: 2 of 3 tests – 14 ms
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
- A user has been added to the list.
java.lang.AssertionError: The username is not correct
Expected :User
Actual :user
<Click to see difference>

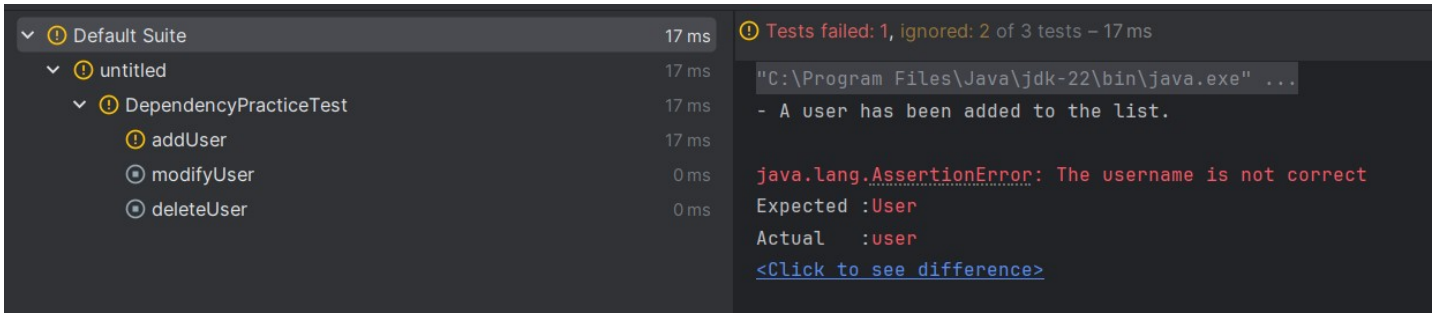
Solution ?

We add : dependsOnMethods ="addUser"

```
package testNGBasics;

import org.testng.Assert;
import org.testng.annotations.Test;

public class DependencyPracticeTest {
      @Test(priority = 0)
      public void addUser() {
            System.out.println("- A user has been added to the list.");
            String expected = "User";
            String actual = "user";
            Assert.assertEquals(actual, expected, "The username is not correct");
```

```
	}


	@Test(priority = 1,  dependsOnMethods ="addUser")

	public void modifyUser() {
		System.out.println("- The user has been successfully modified.");
	}


	@Test(priority = 2,  dependsOnMethods ="addUser")

	public void deleteUser() {
		System.out.println("- The user has been deleted.");
	}
}
```

Result:


Default Suite 17 ms
untitled 17 ms
DependencyPracticeTest 17 ms
addUser 17 ms
modifyUser 0 ms
deleteUser 0 ms
Tests failed: 1, ignored: 2 of 3 tests – 17 ms
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
- A user has been added to the list.
java.lang.AssertionError: The username is not correct
Expected :User
Actual :user
<Click to see difference>

XML

Test Suites

- In any project, you will end up to a place where you need to execute too many test cases on a run.
- Running a set of test cases together is called executing a **Test Suite**.

- We need to create **testng.xml** file to execute multiple test cases as a suite.
- Inside the xml file we specify classes that we want to run as part of the suite.

```
<suite>
 <test name="Test2">
   <classes>
     <class name="MyTestClass1"/>
     <class name="MyTestClass2"/>
     <class name="MyTestClass3"/>
   </classes>
 </test>
</suite>
```

- Run the test by right clicking the testng.xml file and select Run.

In TestNG, XML files play a key role in configuring and executing test suites, providing greater flexibility and control over test execution. Here's a breakdown of the roles and uses of XML files in TestNG:

Key Roles of XML Files in TestNG:

1. **Test Suite Configuration**:
   - XML files allow you to define and organize test suites and test cases. You can group multiple test classes into a single suite, making it easy to manage large test projects.

```
xml
<suite name="Test Suite 1">
    <test name="Test 1">
        <classes>
            <class name="com.example.TestClass1" />
        </classes>
    </test>
</suite>
```

2. **Multiple Test Suites**:
   - You can specify multiple test suites in a single XML file or run multiple XML files. This is useful for categorizing tests based on modules, environments, or other criteria.

```
xml
```

```
<suite name="Master Test Suite">
    <suite-files>
        <suite-file path="suite1.xml" />
        <suite-file path="suite2.xml" />
    </suite-files>
</suite>
```

3. **Test Grouping**:
    - TestNG allows you to group test methods and run specific groups using the XML configuration. This is particularly useful for categorizing tests such as regression, smoke, or integration tests.

```
xml
<groups>
    <run>
        <include name="regression" />
        <exclude name="slow" />
    </run>
</groups>
```

4. **Test Parameterization**:
    - You can define parameters in the XML file and pass them to test methods. This is helpful for running tests with different configurations, such as browser types or user credentials.

```
xml
<suite name="Parameterized Suite">
    <parameter name="browser" value="chrome" />
    <test name="Test with Parameters">
        <classes>
            <class name="com.example.TestClassWithParameters" />
        </classes>
    </test>
</suite>
```

5. **Parallel Execution**:
    - XML files allow you to configure parallel execution for tests or methods. You can run tests in parallel by methods, classes, or suites, speeding up the execution time.

```
xml
<suite name="Parallel Suite" parallel="methods" thread-count="5">
    <test name="Parallel Test">
        <classes>
            <class name="com.example.TestClass" />
        </classes>
    </test>
</suite>
```

6. **Execution Control**:

- You can control the order of test execution by specifying the test method order in the XML file, ensuring certain tests run before others.

```
xml

<suite name="Ordered Test Suite">
    <test name="Ordered Test">
        <classes>
            <class name="com.example.TestClass">
                <methods>
                    <include name="testMethod1" />
                    <include name="testMethod2" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

7. **Suite and Test-Level Annotations**:
   - XML files can define test-level annotations such as `@BeforeSuite`, `@BeforeTest`, `@AfterSuite`, and `@AfterTest`, which help in setting up and tearing down test environments.
8. **Execution Filters**:
   - You can configure filters to run specific methods or groups within classes, using the XML file as the execution blueprint.

Example of a TestNG XML Configuration:

```
xml

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1" parallel="tests" thread-count="2">
    <test name="Test1">
        <classes>
            <class name="com.example.TestClass1"/>
        </classes>
    </test>
    <test name="Test2">
        <classes>
            <class name="com.example.TestClass2"/>
        </classes>
    </test>
</suite>
```

Summary:

- **Test Suite and Test Management**: Organize and execute tests systematically.
- **Grouping and Filtering**: Run specific groups of tests or filter test methods.
- **Parameterization**: Pass dynamic data to test methods.
- **Parallel Execution**: Speed up test runs by executing in parallel.

- **Execution Order Control**: Manage the order of test execution.

XML files in TestNG give you fine-grained control over how your tests are executed, organized, and configured.

Creating xml file:

TEST SUITES

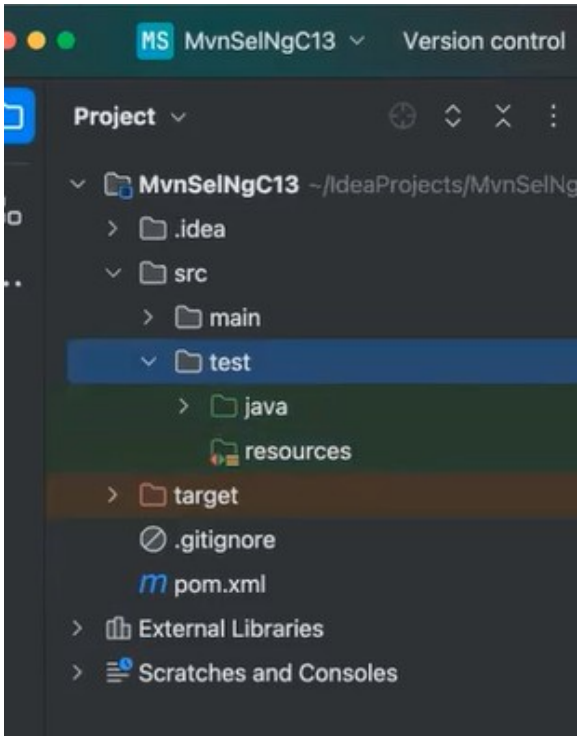
- In any project, you will end up to a place where you need to execute too many test cases on a run.
- Running a set of test cases together is called executing a **Test Suite**.

- We need to create **testng.xml** file to execute multiple test cases as a suite.
- Inside the xml file we specify classes that we want to run as part of the suite.

```
<suite>
 <test name="Test2">
    <classes>
      <class name="MyTestClass1"/>
      <class name="MyTestClass2"/>
      <class name="MyTestClass3"/>
    </classes>
 </test>
</suite>
```

- Run the test by right clicking the testng.xml file and select Run.

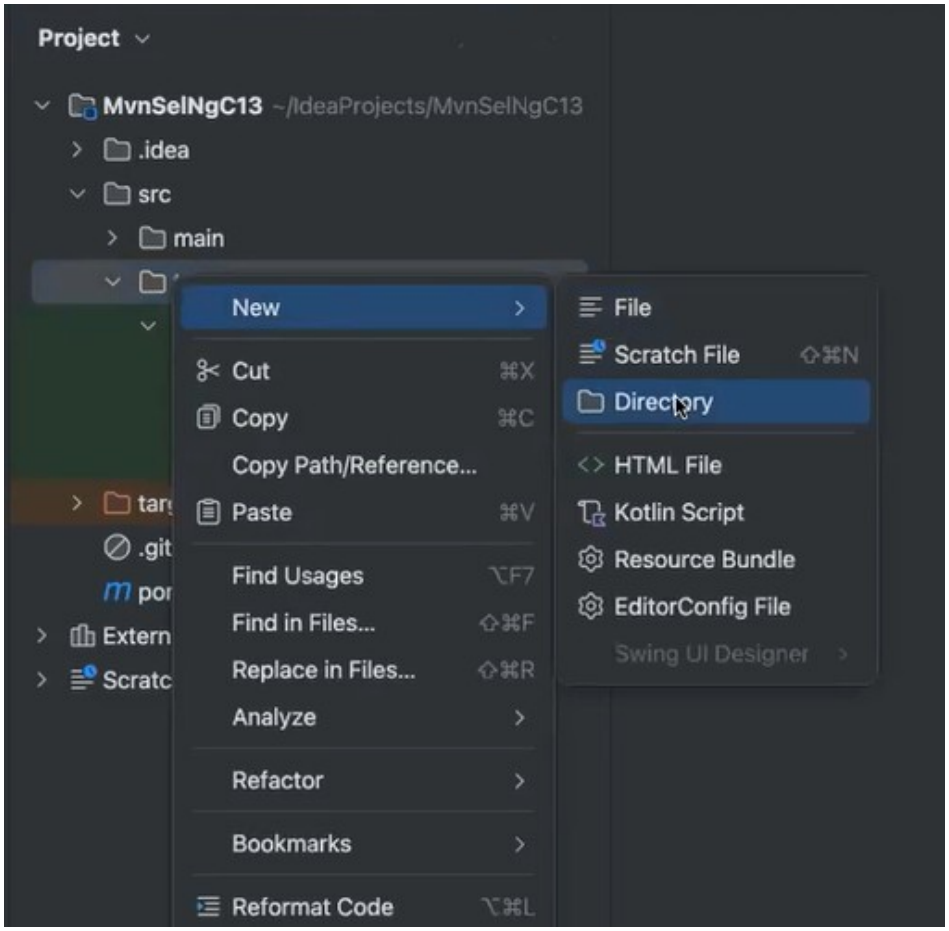
Must be under test package.

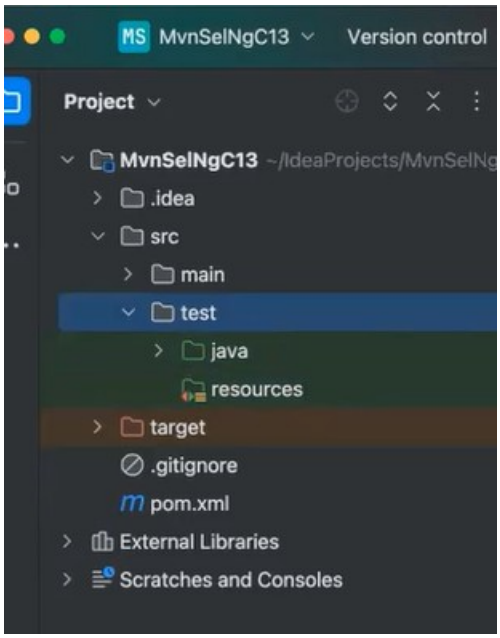


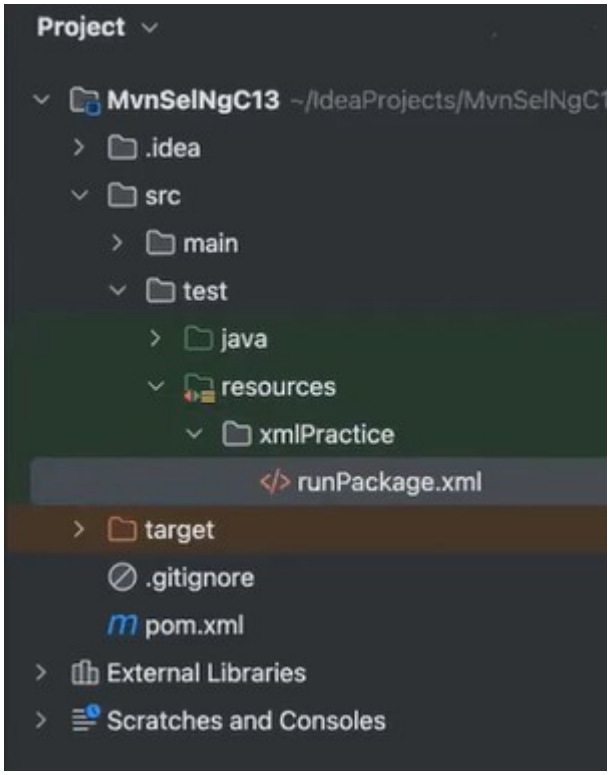
Test/New/ Directory/ name it: ressources

Then create under (ressources) a new directory, name it : xmlPractice

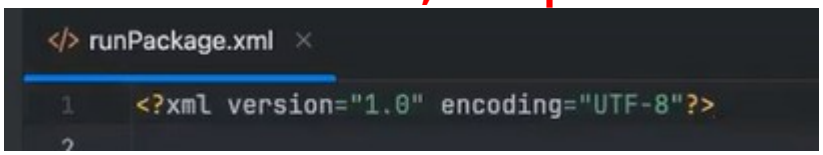
Then (xmlPractice) a new FILE, name it: runPackage.xml


Project
MvnSelNgC13 ~/IdeaProjects/MvnSelNgC13
.idea
src
main
New
Cut ⌘X
Copy ⌘C
Copy Path/Reference...
Paste ⌘V
Find Usages ⌥F7
Find in Files... ⇧⌘F
Replace in Files... ⇧⌘R
Analyze
Refactor
Bookmarks
Reformat Code ⌥⌘L
File
Scratch File ⇧⌘N
Directory
HTML File
Kotlin Script
Resource Bundle
EditorConfig File
Swing UI Designer

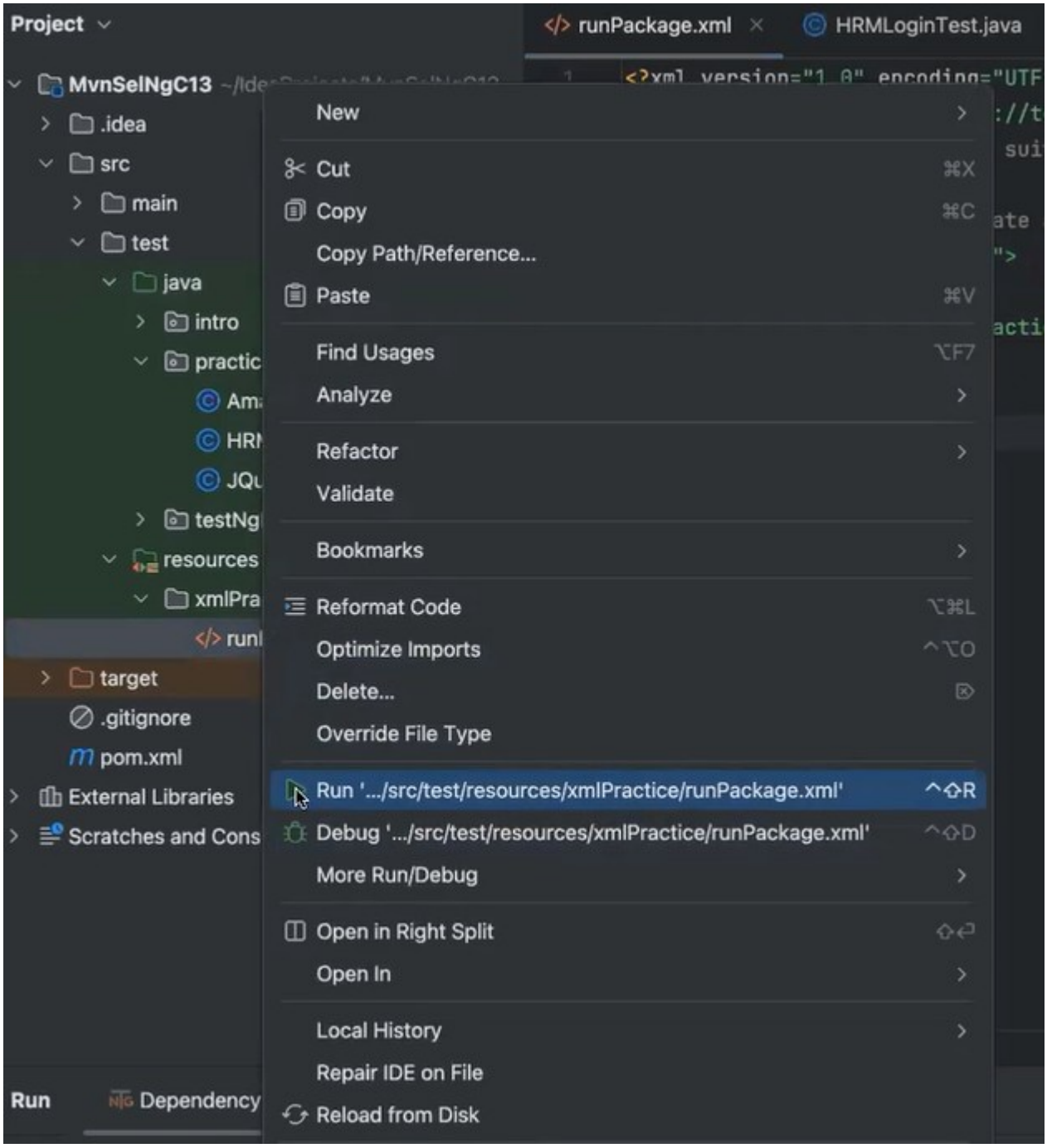

MS MvnSelNgC13
Version control
Project
MvnSelNgC13 ~/IdeaProjects/MvnSelNg
.idea
src
main
test
java
resources
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles



In each xml file, we put this line :

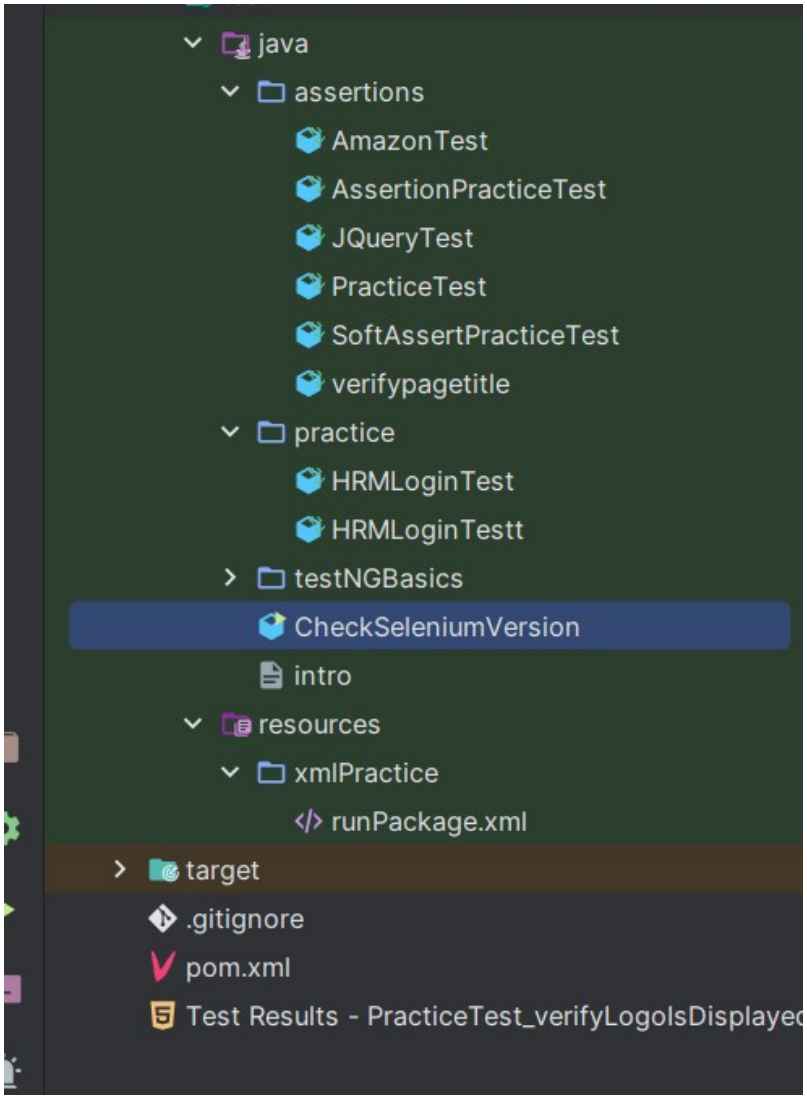


Then run the file



We can run the whole package (**practice**)that contains :

HRMLoginTest Class and HRMLoginTestt Class at the same time by using the xml file: **runPractice.xml**



Save first, then run it by clicking on the xml file each time.

1- Running by package

runPackage.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<!--First we need to create a suite the root tag for testng.xml-->
<suite name="My first Suite">
    <!-- Second we need to create a test-->
    <test name="My first test">
        <packages>
            <package name="practice"/>
```

```
        </packages>
    </test>
</suite>
```

1. **`<?xml version="1.0" encoding="UTF-8"?>`**

This is an XML declaration:

- `version="1.0"`: Specifies the version of XML being used.
- `encoding="UTF-8"`: Defines the character encoding used in the document, which is UTF-8 in this case (a widely used encoding).

2. **`<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">`**

- `<!DOCTYPE>`: This declaration refers to a Document Type Definition (DTD), which defines the structure and rules for the XML document.
- `SYSTEM "https://testng.org/testng-1.0.dtd"`: Specifies the location of the DTD file for TestNG, which helps validate that the XML structure conforms to the rules defined by TestNG. This ensures that the XML file is well-formed and adheres to TestNG's expectations.

3. **`<suite name="My first Suite">`**

- `<suite>`: This is the root tag of the TestNG XML file, representing a test suite. A suite can contain one or more tests.
- `name="My first Suite"`: The `name` attribute defines the name of the suite, which can be any string. This is used for identification purposes when running tests.

4. **`<test name="My first test">`**

- `<test>`: This represents a single test within the suite. Each test can contain multiple classes, packages, or methods.
- `name="My first test"`: The `name` attribute specifies the name of this particular test.

5. **`<packages>`**

- `<packages>`: This tag groups multiple packages together. Packages in Java are namespaces that organize classes and interfaces.

- Inside this tag, you can list one or more `<package>` tags, each specifying a different package.

6. **`<package name="practice"/>`**

- `<package name="practice"/>`: This defines a package named `practice`. It tells TestNG to run all test classes that belong to the `practice` package.

7. **`</suite>`**

- This closes the suite tag and marks the end of the suite definition.

Summary

- **Suite**: The outermost structure (`<suite>`) contains tests.
- **Test**: A test (`<test>`) contains packages or classes with test methods.
- **Packages/Classes**: Inside the test, you specify the packages or individual classes where the test methods are located.

This XML file tells TestNG to execute all test methods inside the `practice` package, grouping them under a test called "My first test" and a suite named "My first Suite."

2- Running by Class

runClass.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<!--First we need to create a suite the root tag for testng.xml-->
<suite name="Run Single Class">
    <!-- Second we need to create a test-->
    <test name="Test Single Class">
        <classes>
            <class name="practice.HRMLoginTestt"/>
            <!-- to Fix this, we put a / at the end of the tag. //lass
name="practice.HRMLoginTest"-->
            <class name="practice.HRMLoginTest"/>

        </classes>
```

```
    </test>
</suite>
```

- **<include>: Runs only the methods listed within the tag.**
- **<exclude>: Runs all methods except the ones listed within the tag.**

3 - Running by Method: INCLUDE

runMethod.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Run a simple test case">
    <test name="Test a simple method">
        <classes>
            <class name="practice.HRMLoginTest">
                <methods>
                    <include name="testSuccessfulLoginDisplaysUsername"/>
                    <include name="testLogoutRedirectsToLoginPage"/>
</methods>
            </class>
            <class name="practice.HRMLoginTestt">
                <methods>
                    <include name="testLogoutRedirectsToLoginPage"/>
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

3 - Running by Method: EXCLUDE

excludeMethod.xml

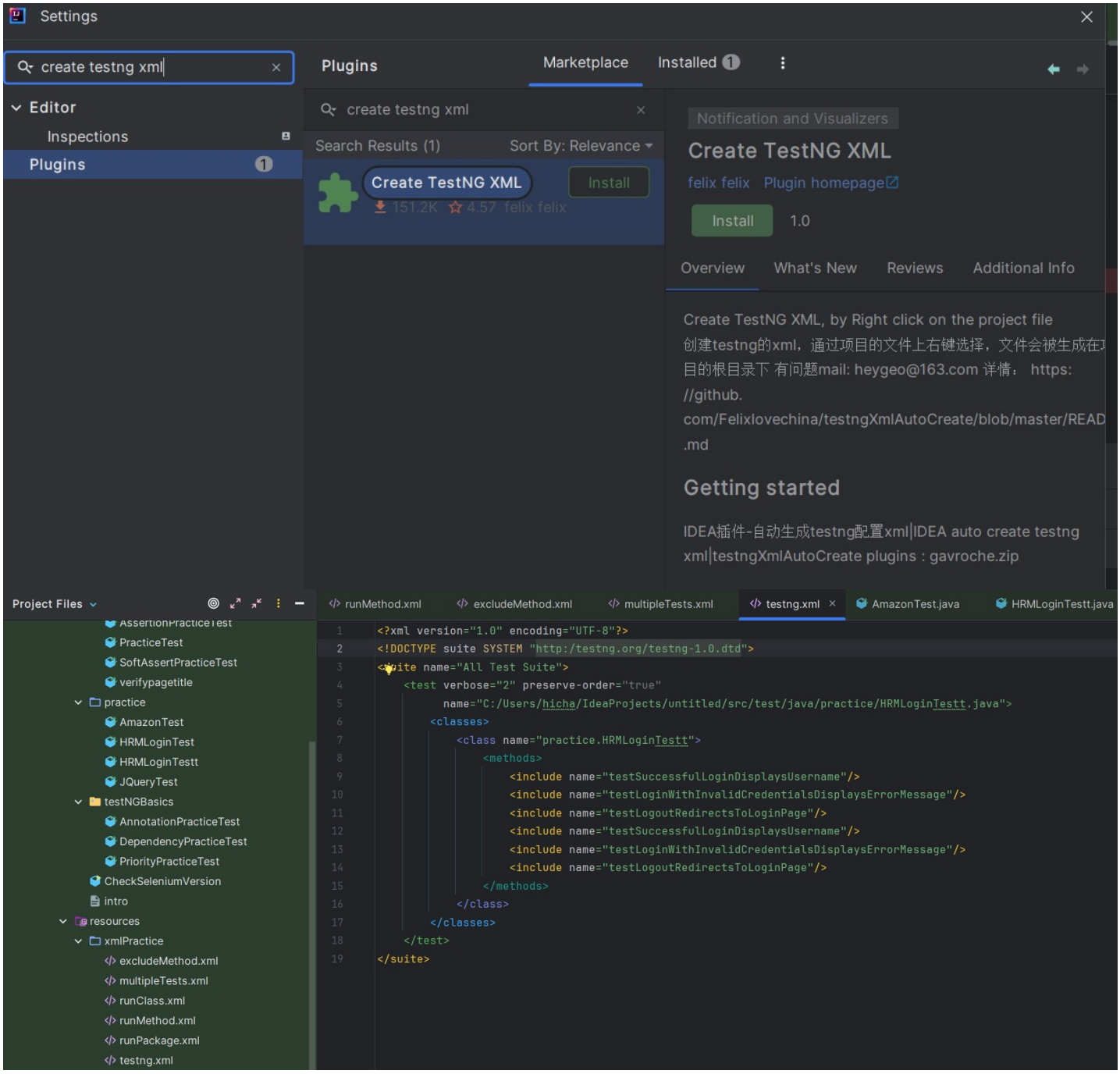
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
```

```
<suite name="Exclude Method">
    <test name="Test exclude method">
        <classes>
            <class name="practice.HRMLoginTestt">
            <methods>
                <exclude
name="testLoginWithInvalidCredentialsDisplaysErrorMessage"/>
                <exclude name="testLogoutRedirectsToLoginPage"/>
            </methods>
            </class>
        </classes>
    </test>
</suite>
```

Create TestNG XML

To create an xml file for a class easily.

1. **Install : Create TestNG XML from Plugins/Marketplace.**
2. **Refresh from the project.**
3. **Click right button on the Class.**
4. **Choose Creat TestNG XML.**
5. **Right Click on the PROJECT/ Reload from Disk.**
6. **Then we can see an xml file created for the class that we had choosen before (Step 3).**
7. **Drag and drop the file to your Directory, and name it.**


Settings
create testng xml
Editor
Inspections
Plugins
Plugins
Marketplace
Installed
create testng xml
Search Results (1)
Sort By: Relevance
Create TestNG XML
Install
151.2K
4.57
felix felix
Notification and Visualizers
Create TestNG XML
felix felix
Plugin homepage
Install
1.0
Overview
What's New
Reviews
Additional Info
Create TestNG XML, by Right click on the project file
创建testng的xml，通过项目的文件上右键选择，文件会被生成在项
目的根目录下 有问题mail: heygeo@163.com 详情： https:
//github.
com/FelixIovechina/testngXmlAutoCreate/blob/master/READ
.md
Getting started
IDEA插件-自动生成testng配置xml|IDEA auto create testng
xml|testngXmlAutoCreate plugins : gavroche.zip
Project Files
PracticeTest
SoftAssertPracticeTest
verifypagetitle
practice
AmazonTest
HRMLoginTest
HRMLoginTestt
JQueryTest
testNGBasics
AnnotationPracticeTest
DependencyPracticeTest
PriorityPracticeTest
CheckSeleniumVersion
intro
resources
xmlPractice
excludeMethod.xml
multipleTests.xml
runClass.xml
runMethod.xml
runPackage.xml
testng.xml
runMethod.xml
excludeMethod.xml
multipleTests.xml
testng.xml
AmazonTest.java
HRMLoginTestt.java
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http:/testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
<test verbose="2" preserve-order="true"
name="C:/Users/hicha/IdeaProjects/untitled/src/test/java/practice/HRMLoginTestt.java">
<classes>
<class name="practice.HRMLoginTestt">
<methods>
<include name="testSuccessfulLoginDisplaysUsername"/>
<include name="testLoginWithInvalidCredentialsDisplaysErrorMessage"/>
<include name="testLogoutRedirectsToLoginPage"/>
<include name="testSuccessfulLoginDisplaysUsername"/>
<include name="testLoginWithInvalidCredentialsDisplaysErrorMessage"/>
<include name="testLogoutRedirectsToLoginPage"/>
</methods>
</class>
</classes>
</test>
</suite>

TestNG

Group of test cases	Test cases
Classes	Methods

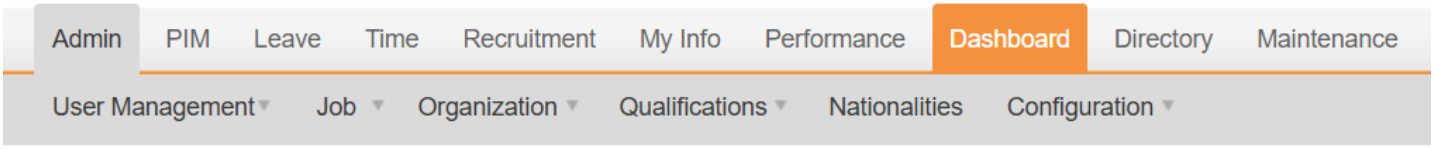
- With TestNG Framework each method is treated as test case
- Classes contain related functionality. Example classes:
  - `Login, Home, Search, ManageUsers`
- Each Class will have related test cases (methods) in it:
  - `test_IsNot_ASenior_If_Age_LessThan_65()`
  - `test_Fail_To_Withdraw_Money_If_Account_Is_Invalid()`
  - `test_Student_Is_Not_Admitted_If_Mandatory_Fields_Are_Missing()`

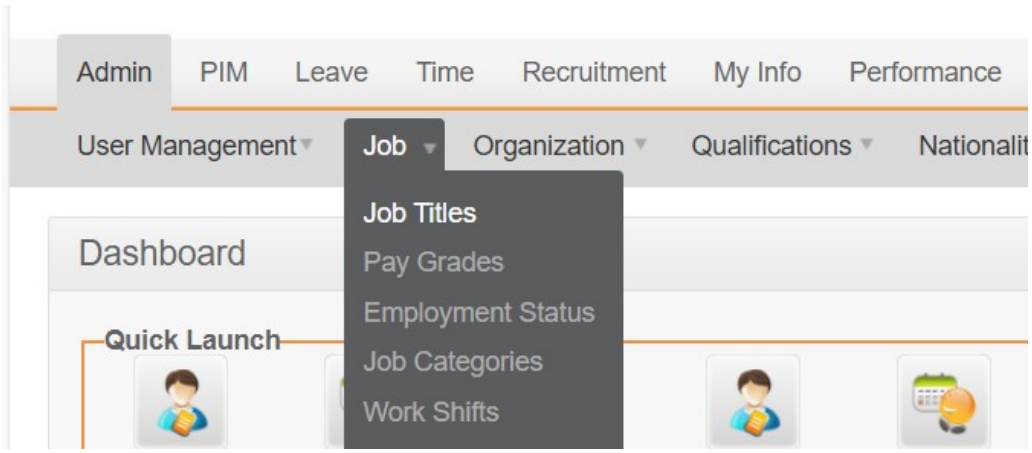
HRM User Story 2 - AC 1

Scenario: User should be able to add Job Title

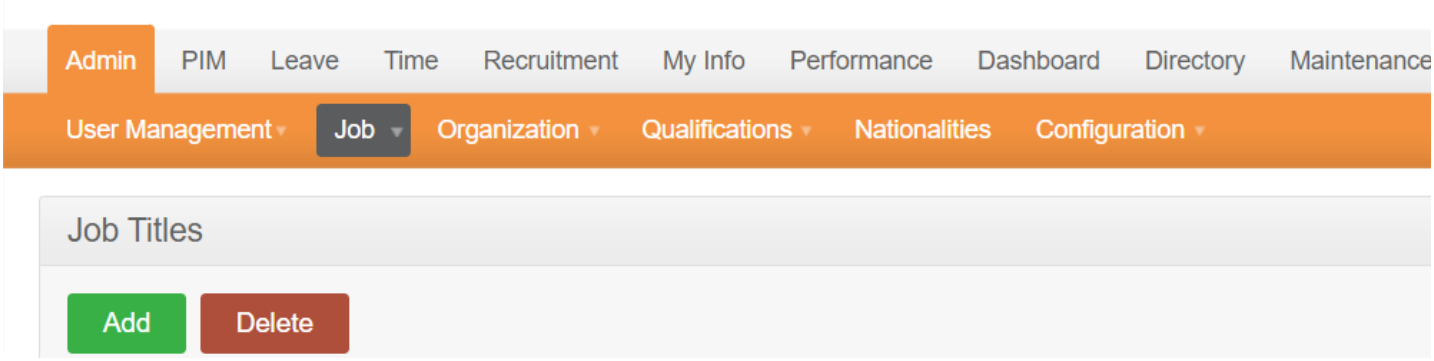
- Given User is on homepage
- User clicks on Admin→ Job → Job Titles
- User can see Add and Delete buttons are displayed
- User can add a new Job Title
- Then newly Added Job Information is displayed in Table

```
// Navigate to Admin > Job > Job Titles
```

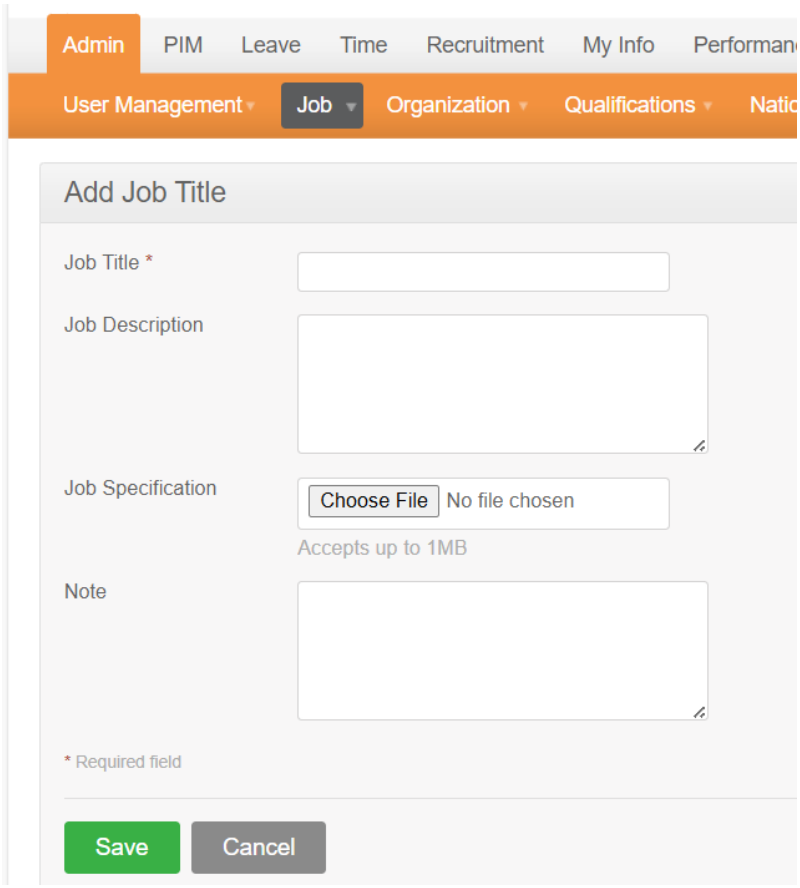




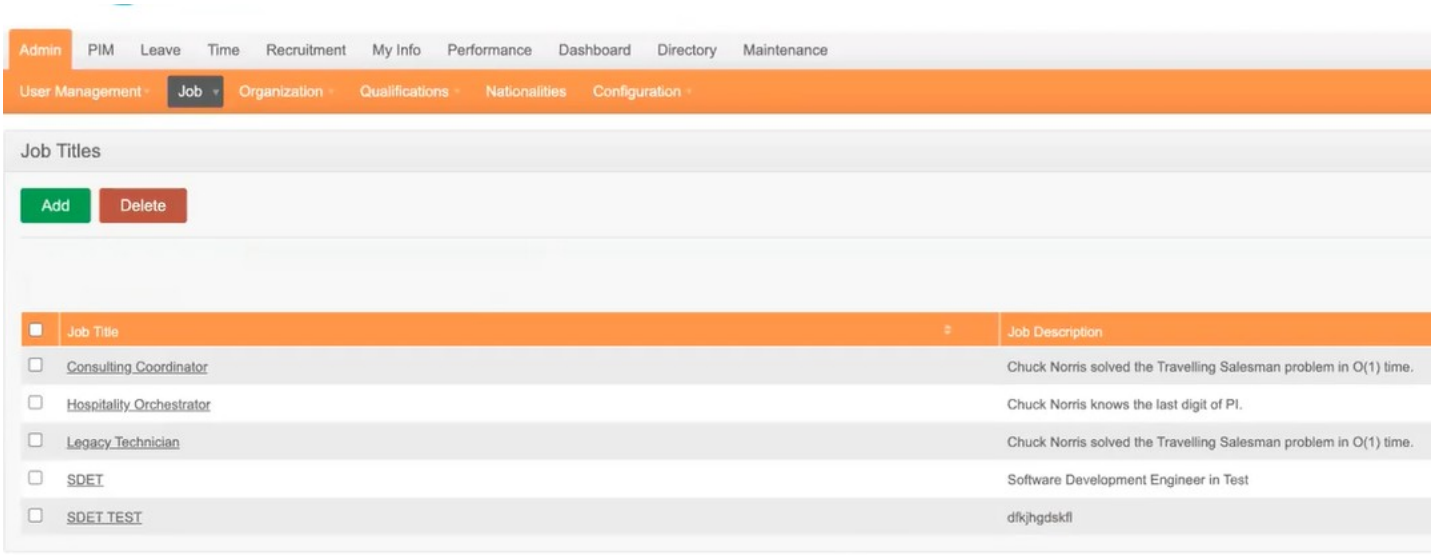

```
// Validate Add and Delete buttons are displayed

// Add new job title
```




```
// Save the job title
```




```
// Verify the newly added job title appears in the table
```



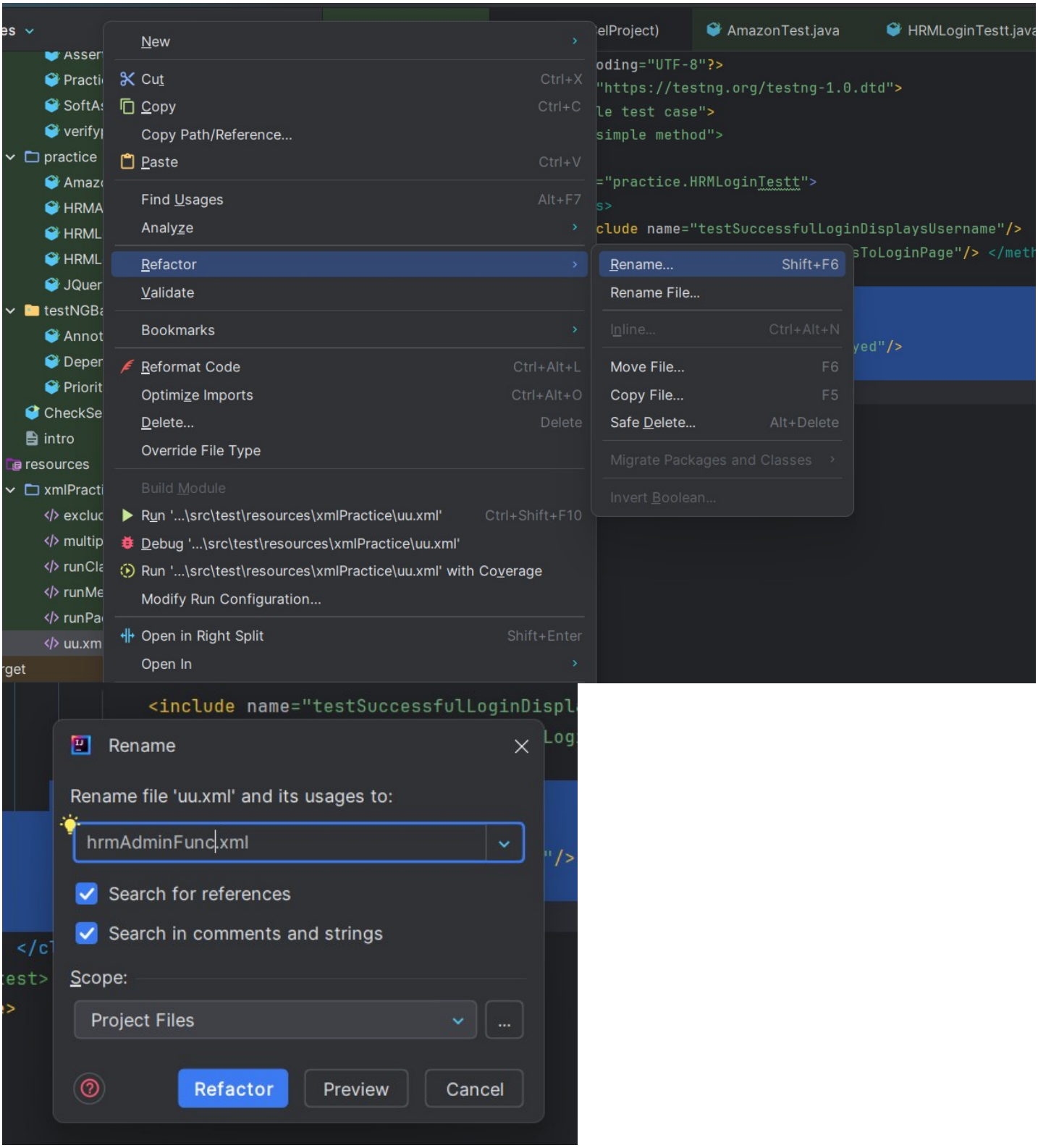

```
/*
In small frameworks we can have one testng.xml config file.
In large frameworks where you have lots of test cases it is common that you create
multiple xml files like:
- regression.xml
- smoke.xml
Option Command + L
Control + Alt + L
 */
package practice;

import com.github.javafaker.Faker;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.util.List;

public class HRMAdminTest {
	private WebDriver driver;

	@BeforeMethod
	public void setUp() {
		driver = new ChromeDriver();
		driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
		driver.manage().window().maximize(); // Maximizing the window for better
interaction
	}

	@AfterMethod
```

```
	public void tearDown() {
		driver.quit();
	}

	/*
	 * Scenario: User should be able to add Job Title Given User is on homepage User
	 * clicks on Admin > Job > Job Titles User can see Add and Delete buttons are
	 * displayed User can add a new Job Title Then newly Added Job Information is
	 * displayed in Table
	 */
	@Test
	public void testUserCanAddAJobTitle() throws InterruptedException {
		Faker mockData = new Faker();

		/* Initialize data to use */
		String username = "yoll-student";
		String password = "Bootcamp5#";
		String jobTitleText = mockData.job().title();
		String jobTitleDescription = mockData.chuckNorris().fact();

		/* Log in */
		login(username, password);

		/* Locate the web elements and Interact with web elements */

		// Navigate to Admin > Job > Job Titles
		WebElement adminTab =
driver.findElement(By.id("menu_admin_viewAdminModule"));
		adminTab.click();
		Thread.sleep(1000);

		WebElement jobDropdown = driver.findElement(By.id("menu_admin_Job"));
		jobDropdown.click();
		Thread.sleep(1000);

		WebElement jobTitleOption =
driver.findElement(By.id("menu_admin_viewJobTitleList"));
		jobTitleOption.click();
		Thread.sleep(1000);

		// Validate Add and Delete buttons are displayed
		WebElement addButton = driver.findElement(By.id("btnAdd"));
		WebElement deleteButton = driver.findElement(By.id("btnDelete"));

		Assert.assertTrue(addButton.isDisplayed(), "Add Button Verification
Failed.");
		Assert.assertTrue(deleteButton.isDisplayed(), "Delete Button Verification
Failed.");

		// Add new job title
		addButton.click();
		WebElement jobTitleInputBox = driver.findElement(By.id("jobTitle_jobTitle"));
		WebElement jobDescriptionInputBox =
driver.findElement(By.id("jobTitle_jobDescription"));
```

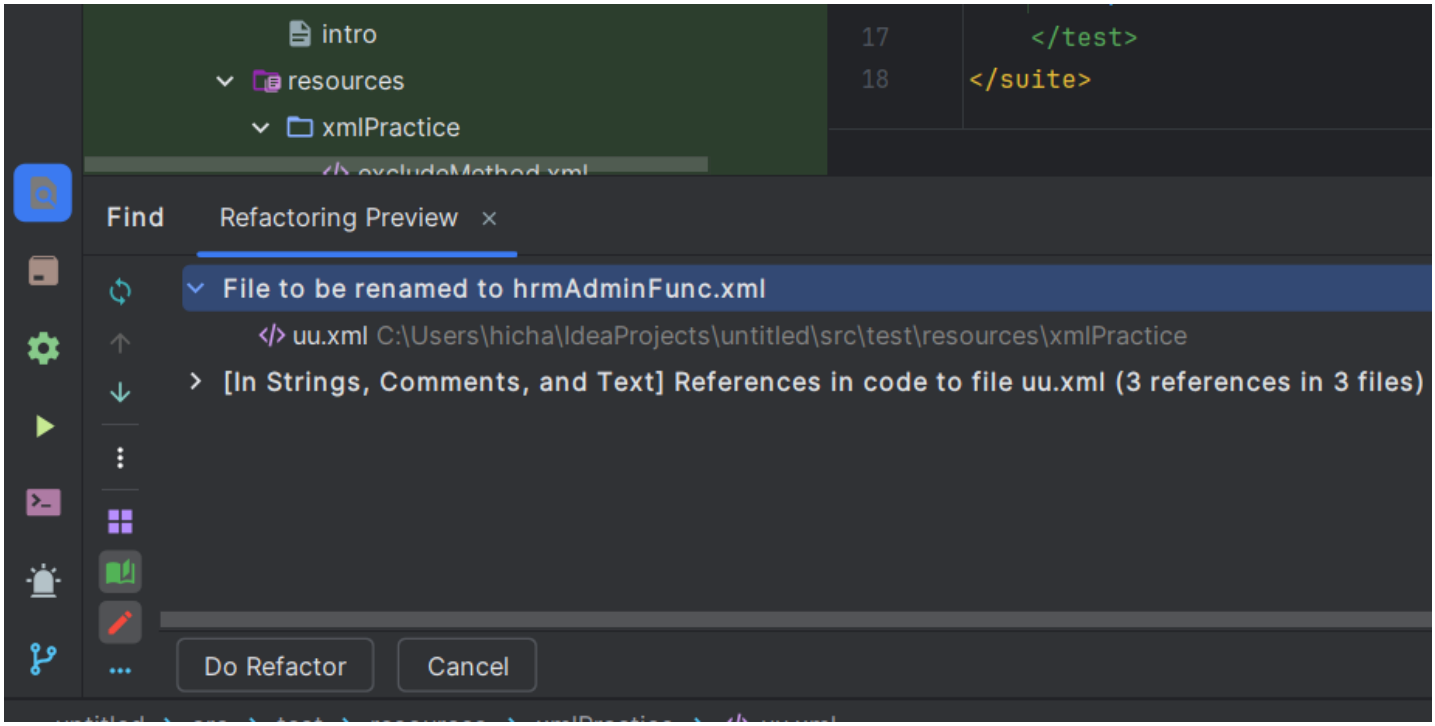
```
            jobTitleInputBox.sendKeys(jobTitleText);
            Thread.sleep(2000);
            jobDescriptionInputBox.sendKeys(jobTitleDescription);
            Thread.sleep(2000);

            // Save the job title
            WebElement saveButton = driver.findElement(By.id("btnSave"));
            saveButton.click();
            Thread.sleep(2000);

            // Verify the newly added job title appears in the table
            List<WebElement> jobTitles =
driver.findElements(By.xpath("//table/tbody/tr/td[2]"));
            boolean jobTitleSuccessfullyAdded = false;

            for (WebElement jobTitle : jobTitles) {
                  if (jobTitle.getText().equals(jobTitleText)) {
                        jobTitleSuccessfullyAdded = true;
                        break;
                  }
            }

            Assert.assertTrue(jobTitleSuccessfullyAdded, "Job title was not successfully
added.");
      }

      // Method to perform login
      public void login(String username, String password) throws InterruptedException {
            WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
            WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
            WebElement loginButton = driver.findElement(By.id("btnLogin"));

            usernameInputBox.sendKeys(username);
            passwordInputBox.sendKeys(password);
            loginButton.click();
            Thread.sleep(2000);
      }
}
```

Creating the xml file for the previous test:

To rename a text file:



Click on Do Refactor



Faker Library

Faker mockData = new Faker(); is a line of code that uses the **Faker** library to create a Faker object named mockData, which can generate random and realistic-looking mock data for testing purposes.

Here's an explanation of each part:

1. **Faker Library:**
   - **Faker** is a popular Java library used to generate fake data, such as names, addresses, phone numbers, emails, dates, and other random information. It helps in creating mock data for testing without needing to hard-code the data, ensuring that your tests are more dynamic and can simulate real-world scenarios.
   - The library is useful for testing applications that require user input, data generation, or randomization. It is especially handy in unit testing, integration testing, or any automated testing scenario.
2. **Creating a Faker Object:**
   - Faker mockData = new Faker();
   - This line creates an instance of the Faker class called mockData. This object can then be used to generate various types of random data through different methods provided by the library.
3. **Usage:**
   - Once the Faker object is created, you can call its methods to generate mock data. For example:

```
String name = mockData.name().fullName(); // Generates a random full name
```

```
String address = mockData.address().fullAddress(); // Generates a random address

String phoneNumber = mockData.phoneNumber().phoneNumber(); // Generates a random phone number
```

- The Faker class contains several categories like name, address, date, internet, finance, and more, each having methods to generate corresponding data.

Example:

```
import com.github.javafaker.Faker;



public class TestFaker {

    public static void main(String[] args) {

        // Create Faker instance

        Faker mockData = new Faker();



        // Generate fake data

        String name = mockData.name().fullName();

        String email = mockData.internet().emailAddress();

        String company = mockData.company().name();

        String phone = mockData.phoneNumber().phoneNumber();



        // Print generated data

        System.out.println("Name: " + name);

        System.out.println("Email: " + email);

        System.out.println("Company: " + company);

        System.out.println("Phone: " + phone);
```

```
    }

}
```

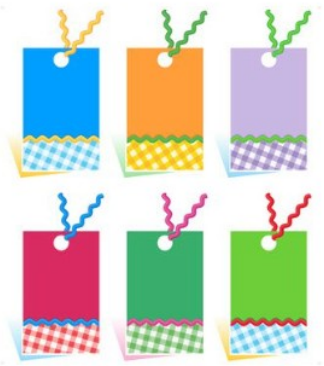
In this example, random fake data such as a full name, email, company, and phone number is generated and printed to the console.

Use Cases:

- **Testing**: It is often used to fill forms, simulate user input, or create mock data for testing APIs, databases, or applications.
- **Dummy Data**: Instead of using hardcoded or repeated data, you can use Faker to generate fresh, random data for each test, making tests more robust.

GROUPING

- **Grouping** → is used to group test cases (**methods**) from different classes into custom groups.
- Can be used to groups like: Smoke, Regression etc.
  - @Test(groups={"smoke"})
  - @Test(groups={"regression"})
  - @Test(groups={"smoke", "regression"})
- Next we select which groups we want to run via the XML file
  -
```
<groups>
    <run>
        <include name="smoke">
    </run>
</<groups>
```
  - If need to run multiple groups just add new include:
    -
```
<include name="smoke">
<include name="regression">
```



TASK:

Test 1, 2, 3, 4, 6, 7 are regression tests.
Test 5 is not a regression test.

regression.xml

```
<?xml_version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Regression Suite">
      <groups>
            <run>
                  <include name="regression" />
```

```
            </run>
      </groups>
      <test name="Regression">
            <classes>
                  <class name="testNGBasics.GroupingPracticeOneTest" />
                  <class name="testNGBasics.GroupingPracticeTwoTest" />
            </classes>
      </test>
</suite>
```

GroupingPracticeOneTest

```
package testNGBasics;

import org.testng.annotations.Test;

public class GroupingPracticeOneTest {

      /*
       * TestNG provides us with a feature called Grouping This allows us to group
       * test by it's purpose regression, smoke
       **/
      @Test(groups = { "regression" })
      public void test_case_one() {
            System.out.println("I am test case ONE!");

      }

      @Test(groups = { "regression" })
      public void test_case_two() {
            System.out.println("I am test case TWO!");

      }

      @Test(groups = { "regression" })
      public void test_case_three() {
            System.out.println("I am test case THREE!");

      }

      @Test(groups = { "regression" })
      public void test_case_four() {
            System.out.println("I am test case FOUR!");
      }

      @Test
      public void test_case_five() {
            System.out.println("I am test case Five!");
      }
}
```

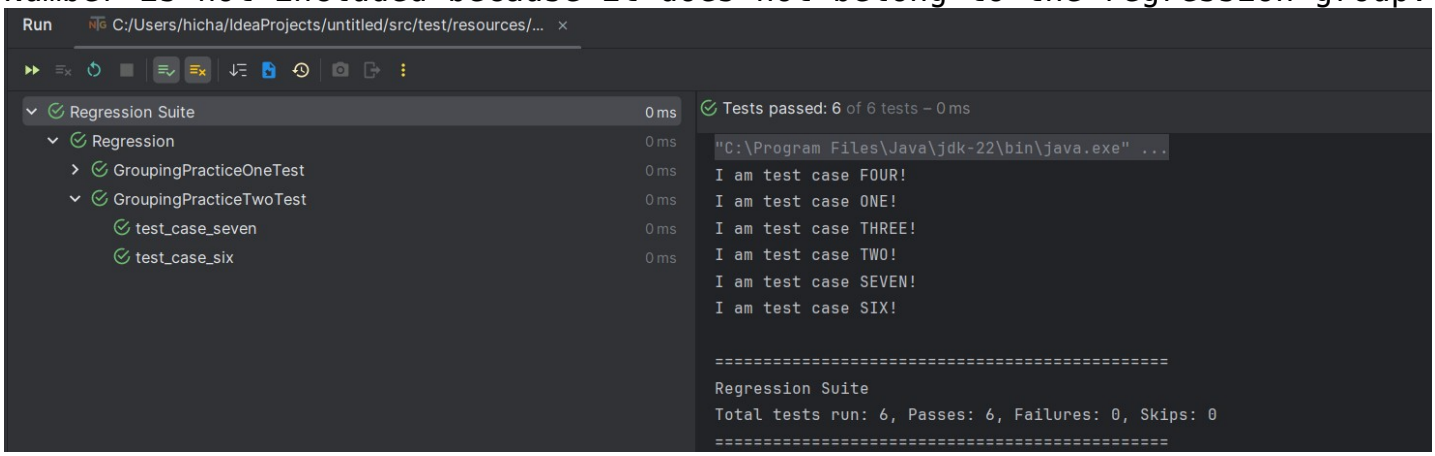
GroupingPracticeTwoTest

```
package testNGBasics;
```

```java
import org.testng.annotations.Test;

public class GroupingPracticeTwoTest {
	@Test(groups = { "regression" })
	public void test_case_six() {
		System.out.println("I am test case SIX!");

	}

	@Test(groups = { "regression" })
	public void test_case_seven() {
		System.out.println("I am test case SEVEN!");

	}

}
```

Result:

Number is not included because it does not belong to the regression group.


Run
C:/Users/hicha/IdeaProjects/untitled/src/test/resources/...
Regression Suite 0 ms
Regression 0 ms
GroupingPracticeOneTest 0 ms
GroupingPracticeTwoTest 0 ms
test_case_seven 0 ms
test_case_six 0 ms
Tests passed: 6 of 6 tests – 0 ms
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
I am test case FOUR!
I am test case ONE!
I am test case THREE!
I am test case TWO!
I am test case SEVEN!
I am test case SIX!
Regression Suite
Total tests run: 6, Passes: 6, Failures: 0, Skips: 0

TASK:

Tests 1, 2, 3, 4, 6, 7 are regression tests.
Test 5 is not a regression test.
Tests 1,2, 6 are smoke tests.

Smoke.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Smoke Suite">
    <groups>
        <run>
            <include name="smoke"/>
        </run>
    </groups>
    <test name="Smoke">
        <packages>
                <package name="testNGBasics"/>
```

```
        </packages>

        <classes>
            <class name="testNGBasics.GroupingPracticeOneTest"/>
            <class name="testNGBasics.GroupingPracticeTwoTest"/>
        </classes>
    </test>
```

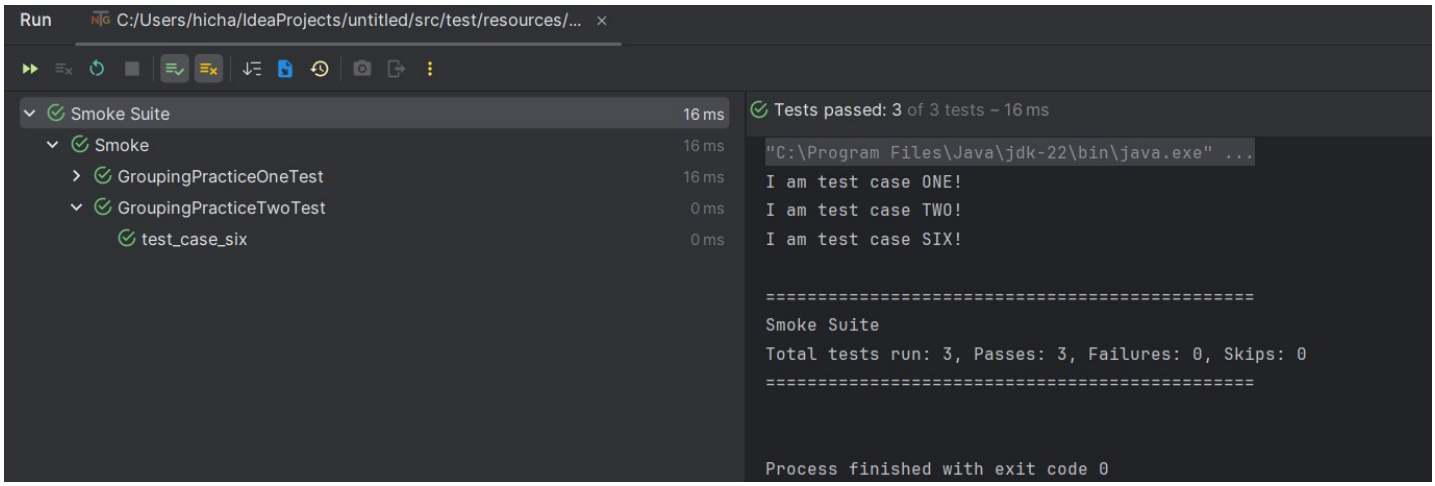
GroupingPracticeOneTest

```
package testNGBasics;

import org.testng.annotations.Test;

public class GroupingPracticeOneTest {

	/*
	 * TestNG provides us with a feature called Grouping This allows us to group
	 * test by it's purpose regression, smoke
	 **/
	@Test(groups = { "regression", "smoke" })
	public void test_case_one() {
		System.out.println("I am test case ONE!");

	}

	@Test(groups = { "regression", "smoke" })
	public void test_case_two() {
		System.out.println("I am test case TWO!");

	}

	@Test(groups = { "regression" })
	public void test_case_three() {
		System.out.println("I am test case THREE!");

	}

	@Test(groups = { "regression" })
	public void test_case_four() {
		System.out.println("I am test case FOUR!");
	}

	@Test
	public void test_case_five() {
		System.out.println("I am test case Five!");
	}
}
```

GroupingPracticeTwoTest

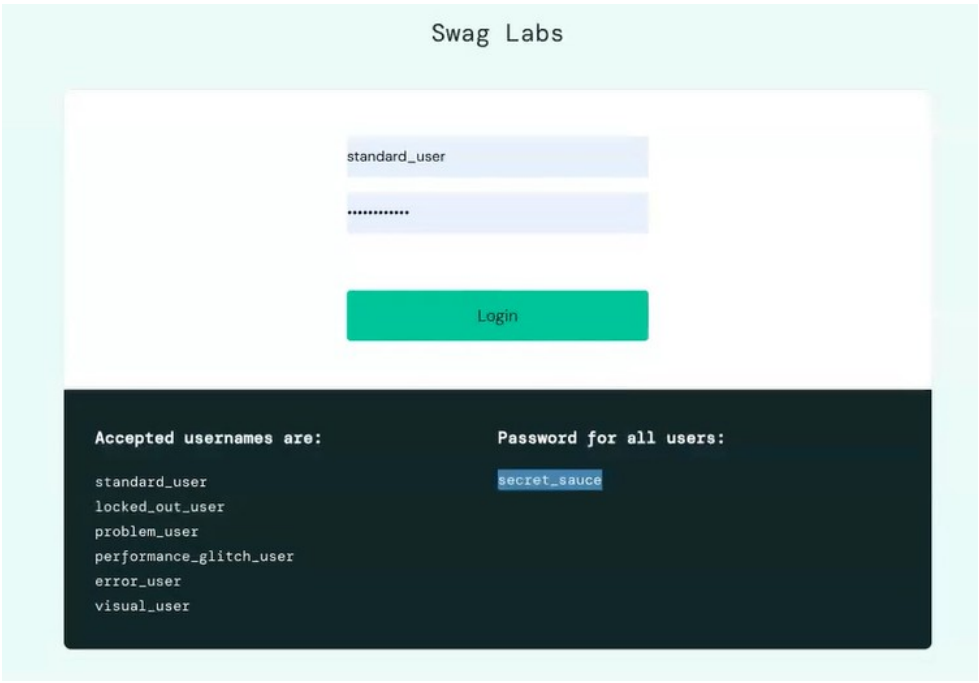
```
package testNGBasics;
```

```java
import org.testng.annotations.Test;

public class GroupingPracticeTwoTest {
	@Test(groups = { "regression", "smoke" })
	public void test_case_six() {
		System.out.println("I am test case SIX!");

	}

	@Test(groups = { "regression" })
	public void test_case_seven() {
		System.out.println("I am test case SEVEN!");

	}

}
```



TASK:

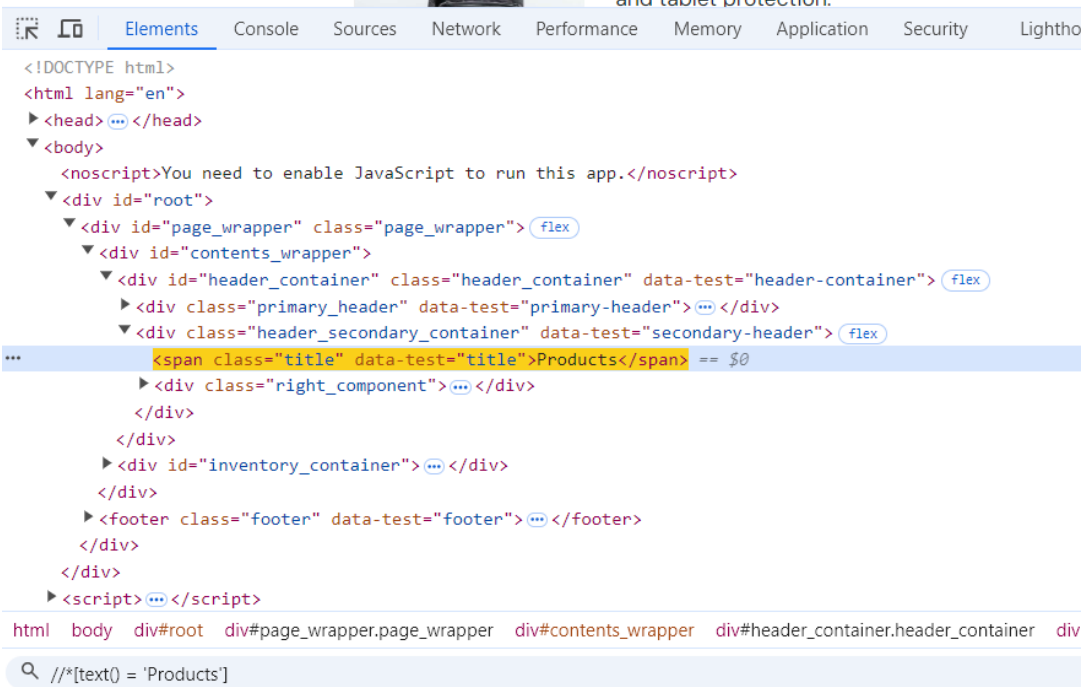
We need to use the same test case with differents data (USERNAMES AND PASSWORDS)



Products

Sauce Labs Backpack

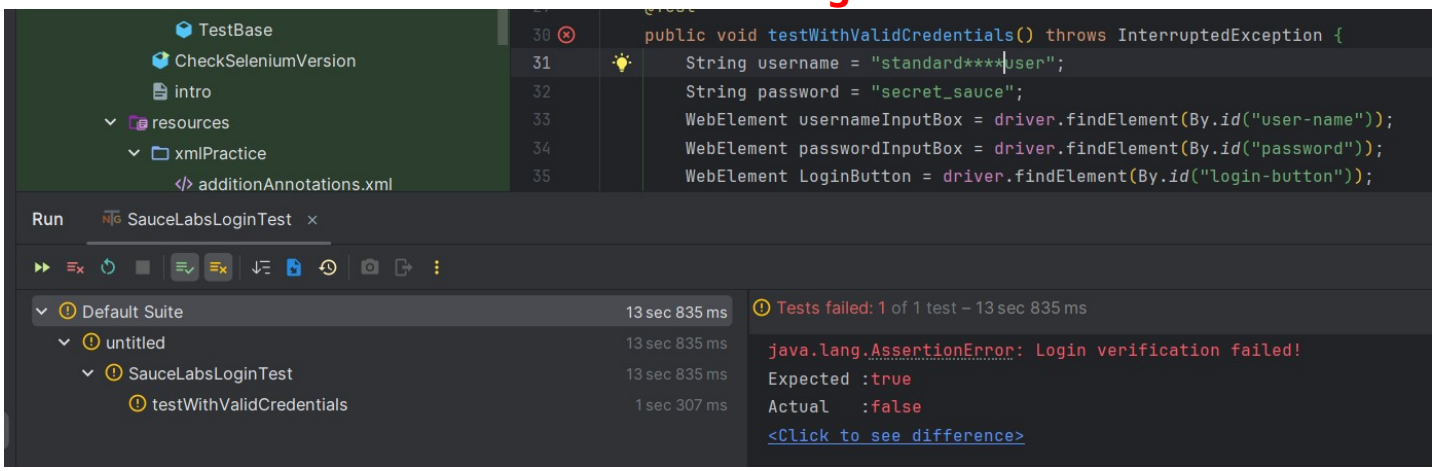
carry.allTheThings() with the sleek, streamlined Sly Pack that melds uncompromising style with unequaled laptop and tablet protection.



Result without failing

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Result with failing : USERNAME is not correct To make sure that the element is existing OR NOT.

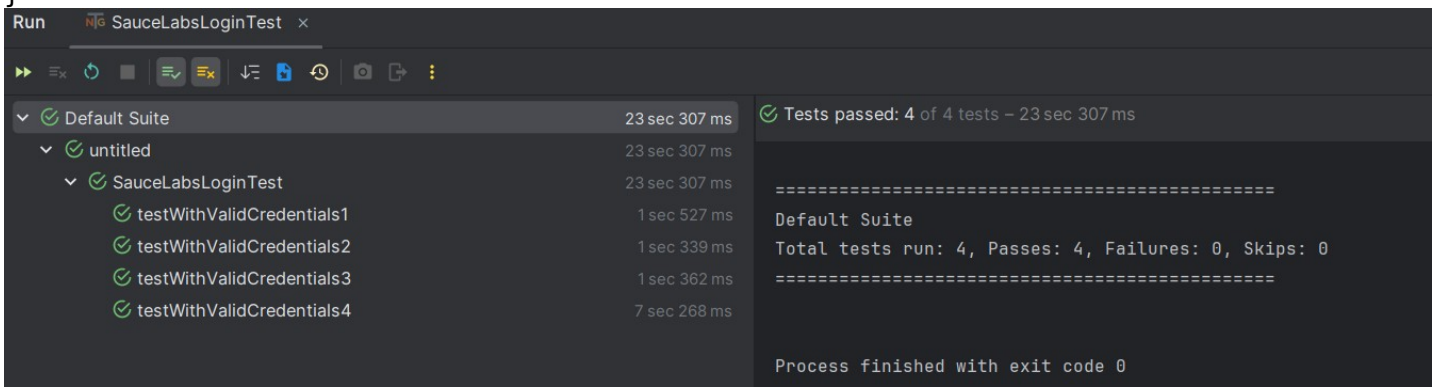


BAD APPROACH: WITH DUPLICATED TESTS CASE

```
package testNGBasics;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class SauceLabsLoginTest {
	private WebDriver driver;

	/* User should be able to successfully login */
	@BeforeMethod
```

```
	public void setUpBrowserNavigate() {

		driver = new ChromeDriver();
// driver.manage().window().maximize();
		driver.get("https://www.saucedemo.com/");
	}

	@AfterMethod
	public void tearDown() {
		driver.quit();
	}

	@Test
	public void testWithValidCredentials1() throws InterruptedException {
		String username = "standard_user";
		String password = "secret_sauce";
		WebElement usernameInputBox = driver.findElement(By.id("user-name"));
		WebElement passwordInputBox = driver.findElement(By.id("password"));
		WebElement LoginButton = driver.findElement(By.id("login-button"));
		usernameInputBox.sendKeys(username);
		passwordInputBox.sendKeys(password);
		LoginButton.click();
		Thread.sleep(1000);
// How to understand if an element is present on the page
// A: Use findElements() and check the size of the list
		boolean isProductLabelPresent = driver.findElements(By.xpath("//*[text() =
'Products']")).size() == 1;
		Assert.assertTrue(isProductLabelPresent, "Login verification failed!");
	}

	@Test
	public void testWithValidCredentials2() throws InterruptedException {
		String username = "visual_user";
		String password = "secret_sauce";
		WebElement usernameInputBox = driver.findElement(By.id("user-name"));
		WebElement passwordInputBox = driver.findElement(By.id("password"));
		WebElement LoginButton = driver.findElement(By.id("login-button"));
		usernameInputBox.sendKeys(username);
		passwordInputBox.sendKeys(password);
		LoginButton.click();
		Thread.sleep(1000);
// How to understand if an element is present on the page
// A: Use findElements() and check the size of the list
		boolean isProductLabelPresent = driver.findElements(By.xpath("//*[text() =
'Products']")).size() == 1;
		Assert.assertTrue(isProductLabelPresent, "Login verification failed!");
	}

	@Test
	public void testWithValidCredentials3() throws InterruptedException {
		String username = "problem_user";
		String password = "secret_sauce";
		WebElement usernameInputBox = driver.findElement(By.id("user-name"));
		WebElement passwordInputBox = driver.findElement(By.id("password"));
		WebElement LoginButton = driver.findElement(By.id("login-button"));
```

```
            usernameInputBox.sendKeys(username);
            passwordInputBox.sendKeys(password);
            LoginButton.click();
            Thread.sleep(1000);
// How to understand if an element is present on the page
// A: Use findElements() and check the size of the list
            boolean isProductLabelPresent = driver.findElements(By.xpath("//*[text() =
'Products']")).size() == 1;
            Assert.assertTrue(isProductLabelPresent, "Login verification failed!");
      }

      @Test
      public void testWithValidCredentials4() throws InterruptedException {
            String username = "performance_glitch_user";
            String password = "secret_sauce";
            WebElement usernameInputBox = driver.findElement(By.id("user-name"));
            WebElement passwordInputBox = driver.findElement(By.id("password"));
            WebElement LoginButton = driver.findElement(By.id("login-button"));
            usernameInputBox.sendKeys(username);
            passwordInputBox.sendKeys(password);
            LoginButton.click();
            Thread.sleep(1000);
// How to understand if an element is present on the page
// A: Use findElements() and check the size of the list
            boolean isProductLabelPresent = driver.findElements(By.xpath("//*[text() =
'Products']")).size() == 1;
            Assert.assertTrue(isProductLabelPresent, "Login verification failed!");
      }

}
```


Run
SauceLabsLoginTest
Default Suite 23 sec 307 ms
untitled 23 sec 307 ms
SauceLabsLoginTest 23 sec 307 ms
testWithValidCredentials1 1 sec 527 ms
testWithValidCredentials2 1 sec 339 ms
testWithValidCredentials3 1 sec 362 ms
testWithValidCredentials4 7 sec 268 ms
Tests passed: 4 of 4 tests – 23 sec 307 ms
Default Suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
Process finished with exit code 0

THE BEST APPROACH:

Parameterized Tests

- This annotation is somewhat similar to @Test annotation.
- The primary purpose of this annotation is to run a test multiple times with different arguments.
- Data Providers are used in TestNG to achieve parameterized tests.

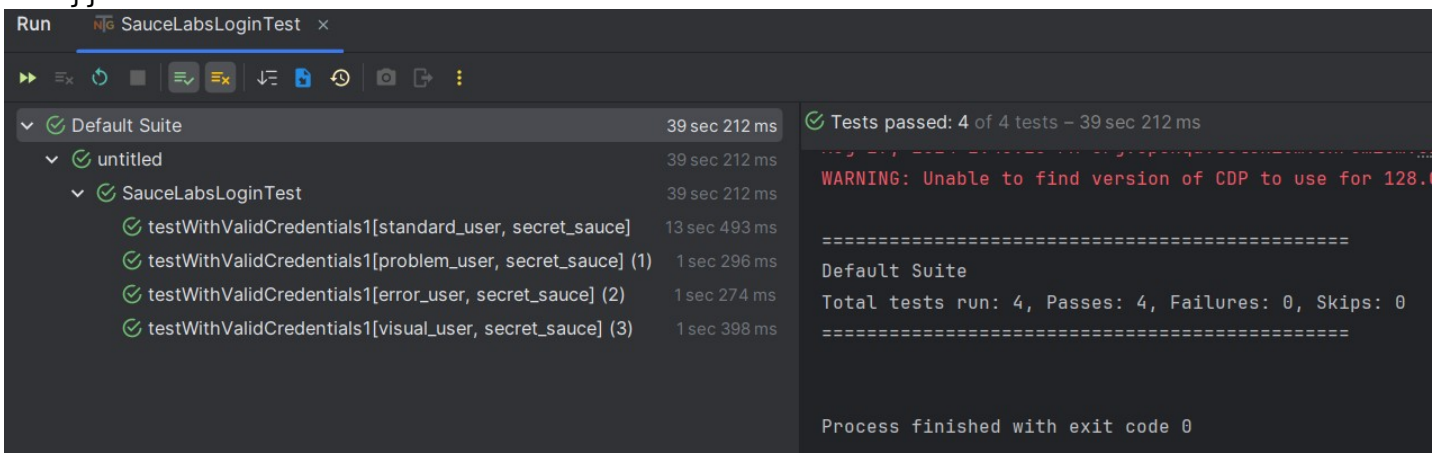
When using Data Providers, TestNG requires that the data is passed in a two-dimensional array, where each row represents a set of input parameters and the columns represent the individual parameters.

```
public class DataProvidersPractice {
    @DataProvider(name="inputs")
    public Object[][] getData() {
        return new Object[][]{
                {"apple", "MacBook"},
                {"microsoft", "SurfaceBook"},
                {"razer", "RazerBook"},
                {"asus", "ZenBook"},
        };}
```

SauceLabsLoginTest

```
package testNGBasics;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class SauceLabsLoginTest {
    private WebDriver driver;

    /* User should be able to successfully login */
    @BeforeMethod
    public void setUpBrowserNavigate() {

        driver = new ChromeDriver();
// driver.manage().window().maximize();
        driver.get("https://www.saucedemo.com/");
    }

    @AfterMethod
    public void tearDown() {
```

```
        driver.quit();
    }

    @Test(dataProvider ="sauceLabsLoginCredentials" )
    public void testWithValidCredentials1(String username,String password ) throws
InterruptedException {

        WebElement usernameInputBox = driver.findElement(By.id("user-name"));
        WebElement passwordInputBox = driver.findElement(By.id("password"));
        WebElement LoginButton = driver.findElement(By.id("login-button"));
        usernameInputBox.sendKeys(username);
        passwordInputBox.sendKeys(password);
        LoginButton.click();
        Thread.sleep(1000);
// How to understand if an element is present on the page
// A: Use findElements() and check the size of the list
        boolean isProductLabelPresent = driver.findElements
                (By.xpath("//*[text() = 'Products']")).size() == 1;
        Assert.assertTrue(isProductLabelPresent, "Login verification failed!");
    }


    // Data Providers
    @DataProvider(name = "sauceLabsLoginCredentials")
    public Object[][] dataSet() {
        Object[][] dataSet = {
                {"standard_user", "secret_sauce"},
                {"problem_user", "secret_sauce"},
                {"error_user", "secret_sauce"},
                {"visual_user", "secret_sauce"}
        };
        return dataSet;

    }}
```


Run
SauceLabsLoginTest
Default Suite 39 sec 212 ms
untitled 39 sec 212 ms
SauceLabsLoginTest 39 sec 212 ms
testWithValidCredentials1[standard_user, secret_sauce] 13 sec 493 ms
testWithValidCredentials1[problem_user, secret_sauce] (1) 1 sec 296 ms
testWithValidCredentials1[error_user, secret_sauce] (2) 1 sec 274 ms
testWithValidCredentials1[visual_user, secret_sauce] (3) 1 sec 398 ms
Tests passed: 4 of 4 tests – 39 sec 212 ms
WARNING: Unable to find version of CDP to use for 128.
Default Suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
Process finished with exit code 0

This code defines a **Data Provider** in TestNG, which is a feature used to supply test methods with data from an external source. This allows you to run the same test multiple times with different sets of data.

Breakdown of the Code:

```
@DataProvider(name = "sauceLabsLoginCredentials")
public Object[][] dataSet() {
    Object[][] dataSet = {
            {"standard_user", "secret_sauce"},
            {"problem_user", "secret_sauce"},
            {"error_user", "secret_sauce"},
            {"visual_user", "secret_sauce"}
    };
    return dataSet;
}
```

Key Concepts:

1. **`@DataProvider` Annotation:**
   - This annotation marks a method as a data provider.
   - The `name` attribute (`"sauceLabsLoginCredentials"`) gives a name to the data provider, which you can refer to in your test methods.
2. **`Object[][] dataSet`:**
   - The data provider returns a 2D array of `Object` type (`Object[][]`).
   - Each row in the 2D array represents a different set of parameters that will be passed to the test method.
   - Each column in the row corresponds to a parameter in the test method.
3. **The Dataset:**
   - The dataset in this example consists of four rows, each representing a pair of login credentials.
   - The first element in each row is the username, and the second is the password.
   - For example, `{"standard_user", "secret_sauce"}` is one set of credentials.
4. **Returning the Dataset:**
   - The `dataSet()` method returns the `Object[][]` array, making this data available to the test methods that use this data provider.

How It Works in a Test:

When you use this data provider in a test method, TestNG will run the test method once for each row in the dataset, passing the elements of the row as parameters to the method.

Example Usage in a Test Method:

```
@Test(dataProvider = "sauceLabsLoginCredentials")
public void loginTest(String username, String password) {
    // Test code that uses the username and password
}
```

- The test method `loginTest` will be executed four times, once for each set of credentials provided by the `sauceLabsLoginCredentials` data provider.
- In each execution, the `username` and `password` parameters will be populated with the corresponding values from one row of the dataset.

Benefits of Using Data Providers:

- **Reusability:** You can reuse the same test logic with different input values.
- **Maintainability:** If you need to update or add test data, you can do it in one place (the data provider method) without modifying the test logic.
- **Scalability:** Easily expand the test to cover more cases by simply adding more rows to the dataset.

Different Tests approaches: USING HRMLoginTestt Class

We can test Login Fuctionality with different approaches:

Positive

- Happy path valid username/password

--Different roles valid credentials

Negative

-Test invalid credentials

-Test empty username

-Test empty password

-Test both empty

-Valid username invalid password

-Invalid username valid password

Based on username and password we will expect a specific message

Boundary

TASK

There are two files:

1. TestData class
2. HRMLoginTestt class

TestData class

```
package practice;

import org.testng.annotations.DataProvider;

public class TestData {
	@DataProvider(name = "hrmLoginDataSet")
	public Object[][] getTestData() {
		return new Object[][] {
				{ "invalid", "invalid", "Invalid credentials" },
				{ "", "invalid", "Username cannot be empty" },
				{ "invalid", "", "Password cannot be empty" },
				{ "", "", "Username cannot be empty" }
				};
	}
}
```

Explanation of `TestData class`

The `TestData` class in the provided code is designed to supply data to the test methods in your TestNG framework. It contains a `DataProvider` method named `getTestData`, which is used to provide different sets of input data to the test methods that are marked with the `@Test` annotation. Here's a breakdown of the code:

1. Package and Imports:

- The class is part of the `practice` package.
- It uses `org.testng.annotations.DataProvider` from the TestNG library to create a data provider.

2. `@DataProvider` Annotation:

- The `@DataProvider` annotation is used to mark a method as a data provider in TestNG. This method provides the data that will be used by the test methods.
- The `name` attribute is used to give a name to the data provider, which can be referenced in test methods. In this case, the data provider is named `"hrmLoginDataSet"`.

3. `getTestData` Method:

- This method returns a 2-dimensional array of `Object[][]` type.
- Each inner array represents a set of test data (input parameters) that will be passed to the test method.

4. Data Provided:

- The method provides four sets of login data, each consisting of three elements: a `username`, a `password`, and an `expectedErrorMessage`.
- **Data Sets:**
  1. `{"invalid", "invalid", "Invalid credentials"}`:
     - This data set tests the scenario where both the username and password are invalid. The expected result is an error message `"Invalid credentials"`.
  2. `{"", "invalid", "Username cannot be empty"}`:
     - This data set tests the scenario where the username is empty and the password is invalid. The expected result is an error message `"Username cannot be empty"`.
  3. `{"invalid", "", "Password cannot be empty"}`:
     - This data set tests the scenario where the username is invalid and the password is empty. The expected result is an error message `"Password cannot be empty"`.
  4. `{"", "", "Username cannot be empty"}`:
     - This data set tests the scenario where both the username and password are empty. The expected result is an error message `"Username cannot be empty"`.

5. Usage in Test Methods:

- This data provider can be used in a test method by specifying the `dataProvider` attribute in the `@Test` annotation, like this:

```
@Test(dataProvider = "hrmLoginDataSet", dataProviderClass = TestData.class)
public void testLoginWithInvalidCredentialsDisplaysErrorMessage(String username,
String password, String expectedErrorMessage) {
    // Test code here
}
```

- The test method will be executed multiple times, once for each set of data provided by the `getTestData` method.

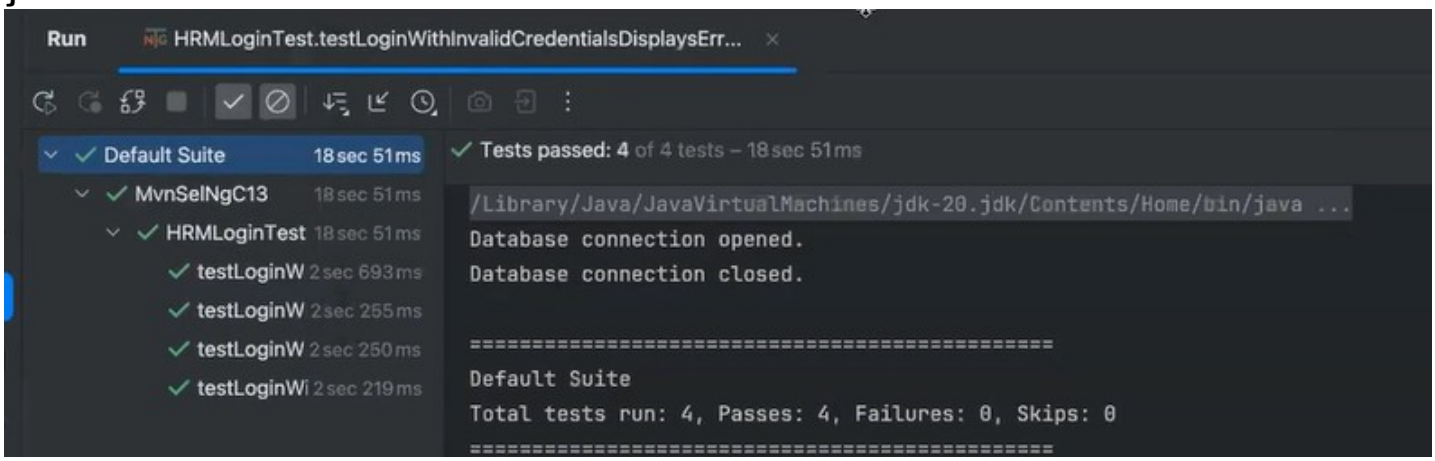
Purpose:

- The purpose of creating a `DataProvider` is to facilitate data-driven testing. This allows you to run the same test method multiple times with different sets of data, making your tests more robust and reducing redundancy in test code.
- By using a `DataProvider`, you can separate the test logic from the test data, making your tests easier to maintain and extend.

In summary, the `TestData` class is a utility for supplying multiple sets of test data to test methods in a TestNG framework. It enhances the flexibility and scalability of your test suite by allowing you to run the same test logic against various input data sets.

In HRMLoginTestt class

```
@Test(dataProviderClass = TestData.class, dataProvider = "hrmLoginDataSet")
    public void testLoginWithInvalidCredentialsDisplaysErrorMessage(String
username, String password, String expectedErrorMessage) throws
InterruptedException{
    login (username,password);
    WebElement errorMessageElement = driver.findElement(By.id("spanMessage"));
    String actualErrorMessage = errorMessageElement.getText();
    Assert.assertEquals(actualErrorMessage,expectedErrorMessage,"Error message
verification failed.");
}
```


Run
HRMLoginTest.testLoginWithInvalidCredentialsDisplaysErr...
Default Suite 18 sec 51 ms
MvnSelNgC13 18 sec 51 ms
HRMLoginTest 18 sec 51 ms
testLoginW 2 sec 693 ms
testLoginW 2 sec 255 ms
testLoginW 2 sec 250 ms
testLoginWi 2 sec 219 ms
Tests passed: 4 of 4 tests – 18 sec 51 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Database connection opened.
Database connection closed.
Default Suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0

All Tests Cases

Scenario: As an admin I should be able to successfully add Employee

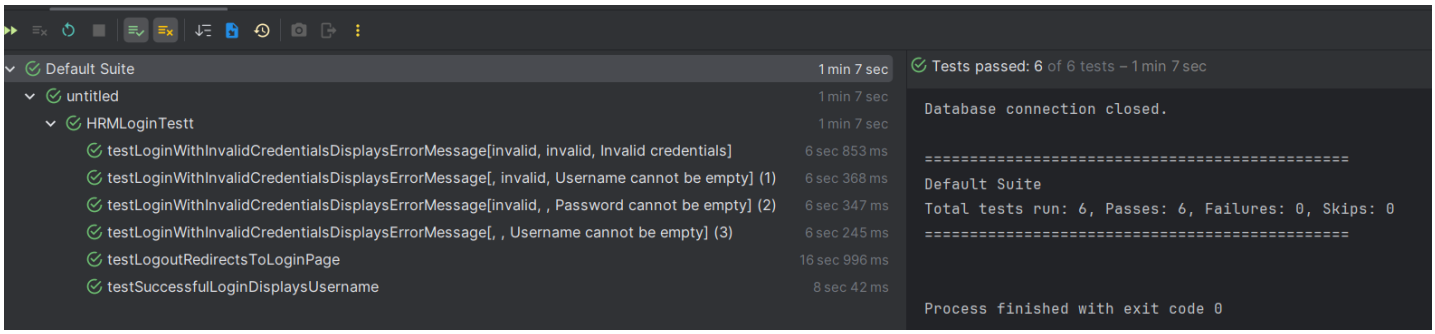
- Given I'm logged in to HRM with Admin Account
- And I click on PIM → Add Employee
- Then I fill out First Name, Middle Name, Last Name
- And I verify Employee number is not empty
- Then I click on Save
- And I verify new account got created with correct details

```java
package practice;

import com.github.javafaker.Faker;
import jdk.jfr.Enabled;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.*;

public class HRMLoginTestt {

	private WebDriver driver;

	/*
	 * If we want to disable a test case not execute - comment it out - remove it -
	 * TestNG provides you a property that can disable a method - * To disable a
	 * method we mark it as enabled = false By default all test are enabled = true
	 **/

	@BeforeClass
	public void dbSetUpConnection() {
		System.out.println("Database connection opened.");
	}

	/*
	 * Summary JUnit: @BeforeClass and @AfterClass must be static
	 * TestNG: @BeforeClass and @AfterClass do not need to be static. BE CAREFUL
	 * WHEN CHOOSING THE ANNOTATIONS, IT MUST BE FOR TestNG NOT JUnit !
	 */
	@AfterClass
	public void dbCleanUp() {
		System.out.println("Database connection closed.");
	}

	@BeforeMethod
	public void setUpBrowserNavigate() {
		driver = new ChromeDriver();
		driver.manage().window().maximize();
		driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
	}
```

```
	@AfterMethod
	public void tearDown() {
		driver.quit();
	}

	@Test(enabled = true)
	public void testSuccessfulLoginDisplaysUsername() throws InterruptedException {
		login("yoll-student", "Bootcamp5#");

		// Validate the welcome message
		WebElement usernameMessage = driver.findElement(By.id("welcome"));
		String expectedUsername = "Welcome Yoll";
		String actualUsername = usernameMessage.getText();
		Assert.assertEquals(expectedUsername, actualUsername, "Username verification
failed.");
	}
@Test(dataProviderClass = TestData.class, dataProvider = "hrmLoginDataSet")
	public void testLoginWithInvalidCredentialsDisplaysErrorMessage(String
username, String password,
			String expectedErrorMessage) throws InterruptedException {
		login(username, password);
		WebElement errorMessageElement =
driver.findElement(By.id("spanMessage"));
		String actualErrorMessage = errorMessageElement.getText();
		Assert.assertEquals(actualErrorMessage, expectedErrorMessage, "Error
message verification failed.");
	}


	@Test
	public void testLogoutRedirectsToLoginPage() throws InterruptedException {
		login("yoll-student", "Bootcamp5#");

		// Locate and click on the dropdown menu
		WebElement dropdownMenu = driver.findElement(By.id("welcome"));
		dropdownMenu.click();

		// Wait for the logout option to be available and click it
		Thread.sleep(5000);
		WebElement logoutOption =
driver.findElement(By.xpath("//a[text()='Logout']"));
		logoutOption.click();

		// Wait for the logout to complete
		Thread.sleep(2000);

		// Validate if the login page is displayed
		WebElement loginHeaderPanel = driver.findElement(By.id("logInPanelHeading"));
		Assert.assertTrue(loginHeaderPanel.isDisplayed(), "Login panel verification
failed.");
	}

	public void login(String username, String password) throws InterruptedException {
		WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
```

```
        WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
        WebElement loginButton = driver.findElement(By.id("btnLogin"));

        usernameInputBox.sendKeys(username);
        Thread.sleep(2000);
        passwordInputBox.sendKeys(password);
        Thread.sleep(2000);
        loginButton.click();
        Thread.sleep(2000);
}
/*
 * Scenario: As an admin I should be able to successfully add Employee Given I'm
 * logged in to HRM with Admin Account And I click on PIM Add Employee Then I
 * fill out First Name, Middle Name, Last Name And I verify Employee number is
 * not empty Then I click on Save And I verify new account got created with
 * correct details
 */

@Test
public void testUserCanAddAnEmployee() throws InterruptedException {
        /* Login to HRM */
        String username = "yoll-student";
        String password = "Bootcamp5#";
        login(username, password);
        /* Navigate to Add Employee Page */
        WebElement pimTab = driver.findElement(By.id("menu_pim_viewPimModule"));
        pimTab.click();
        Thread.sleep(1000);
        WebElement addButton = driver.findElement(By.id("btnAdd"));
        addButton.click();
        Thread.sleep(1000);
        WebElement firstNameInputBox = driver.findElement(By.id("firstName"));
        WebElement middleNameInputBox = driver.findElement(By.id("middleName"));
        WebElement lastNameInputBox = driver.findElement(By.id("lastName"));

        Faker faker = new Faker();
        String firstName = faker.name().firstName();
        String middleName = faker.funnyName().name();
        String LastName = faker.name().lastName();

        firstNameInputBox.sendKeys(firstName);
        Thread.sleep(1000);
        middleNameInputBox.sendKeys(middleName);
        Thread.sleep(1000);
        lastNameInputBox.sendKeys(LastName);
        Thread.sleep(1000);

        WebElement idField = driver.findElement(By.id("employeeId"));

        /* Option 1 */
        String fillingId = idField.getAttribute("value");
        boolean isNotEmpty = !fillingId.isEmpty();
        Assert.assertTrue(isNotEmpty, "Id field verification failed.");
        /* Option 2 */
        // Assert.assertFalse (idField.getAttribute("value").isEmpty(), "Id field
```

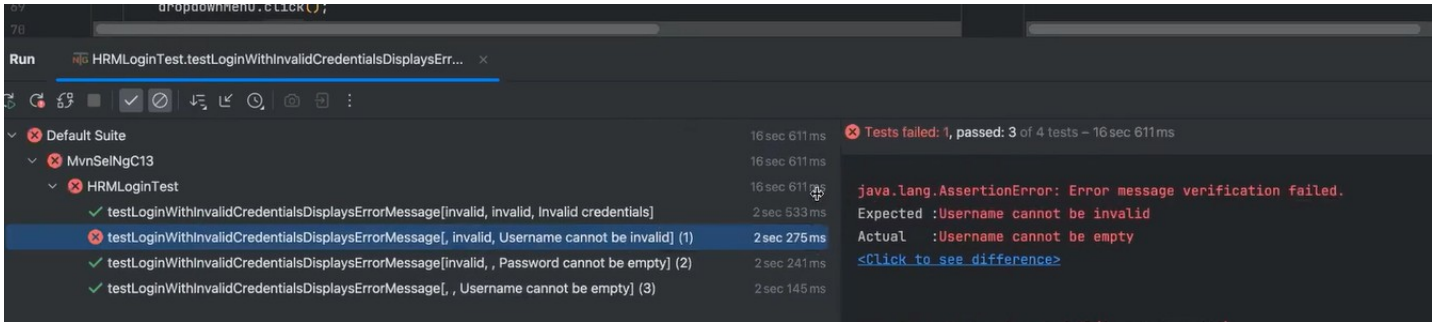
```
            // verification failed.");
            WebElement saveButton = driver.findElement(By.id("btnSave"));
            saveButton.click();
            /* Extract the personal information after the employee was created */
            WebElement actualFirstName =
driver.findElement(By.id("personal_txtEmpFirstName"));
            WebElement actualMiddleName =
driver.findElement(By.id("personal_txtEmpMiddleName"));
            WebElement actualLastName =
driver.findElement(By.id("personal_txtEmpLastName"));
            WebElement actualId = driver.findElement(By.id("personal_txtEmployeeId"));

            /* Assert personal Details */
            Assert.assertEquals(actualFirstName.getAttribute("value"), firstName, "First
name verification failed");
            Assert.assertEquals(actualMiddleName.getAttribute("value"), middleName,
"Middle name verification failed");
            Assert.assertEquals(actualLastName.getAttribute("value"), LastName, "Last
name verification failed");

            Assert.assertEquals(actualId.getAttribute("value"), fillingId, "Id
verification failed");
      }
}
```


Default Suite 1 min 7 sec
untitled 1 min 7 sec
HRMLoginTestt 1 min 7 sec
testLoginWithInvalidCredentialsDisplaysErrorMessage[invalid, invalid, Invalid credentials] 6 sec 853 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[, invalid, Username cannot be empty] (1) 6 sec 368 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[invalid, , Password cannot be empty] (2) 6 sec 347 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[, , Username cannot be empty] (3) 6 sec 245 ms
testLogoutRedirectsToLoginPage 16 sec 996 ms
testSuccessfulLoginDisplaysUsername 8 sec 42 ms
Tests passed: 6 of 6 tests – 1 min 7 sec
Database connection closed.
Default Suite
Total tests run: 6, Passes: 6, Failures: 0, Skips: 0
Process finished with exit code 0

If the expected error message # Actual error message

```
@DataProvider(name = "hrmLoginDataSet")
public Object[][] getTestData(){
    return new Object[][] {
            {"invalid", "invalid", "Invalid credentials"},
            {"",        "invalid", "Username cannot be invalid"},
            {"invalid",        "", "Password cannot be empty"},
            {"",               "", "Username cannot be empty"}
    };
}
```

Result:


dropdownMenu.click();
Run
HRMLoginTest.testLoginWithInvalidCredentialsDisplaysErr...
Default Suite 16 sec 611 ms
MvnSelNgC13 16 sec 611 ms
HRMLoginTest 16 sec 611 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[invalid, invalid, Invalid credentials] 2 sec 533 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[, invalid, Username cannot be invalid] (1) 2 sec 275 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[invalid, , Password cannot be empty] (2) 2 sec 241 ms
testLoginWithInvalidCredentialsDisplaysErrorMessage[, , Username cannot be empty] (3) 2 sec 145 ms
Tests failed: 1, passed: 3 of 4 tests – 16 sec 611 ms
java.lang.AssertionError: Error message verification failed.
Expected :Username cannot be invalid
Actual :Username cannot be empty
<Click to see difference>

PARAMETERS FROM TESTNG.XML

- If you are using simple values for your parameters, you can specify them in your testng.xml:

```
@Parameters({"env"})
@Test
public void dataFromXMLEnv(String env){
   String baseUrl = "bestBuy.com";
   String envUrl = env + "."+ baseUrl;
   System.out.println(envUrl);
}
```

In this code, we specify that the parameter env of your Java method should receive the value of the XML parameter called env. This XML parameter is defined in testng.xml:

```
<suite name="Practice">
   <parameter name="env" value="dev"></parameter>
   <test name="bestBuy">
      <classes>
         <class name="practice.parametrrization.MorePracticeWithParametrizationTest"></class>
      </classes>
   </test>
</suite
```

The same technique can be used for @Before/After

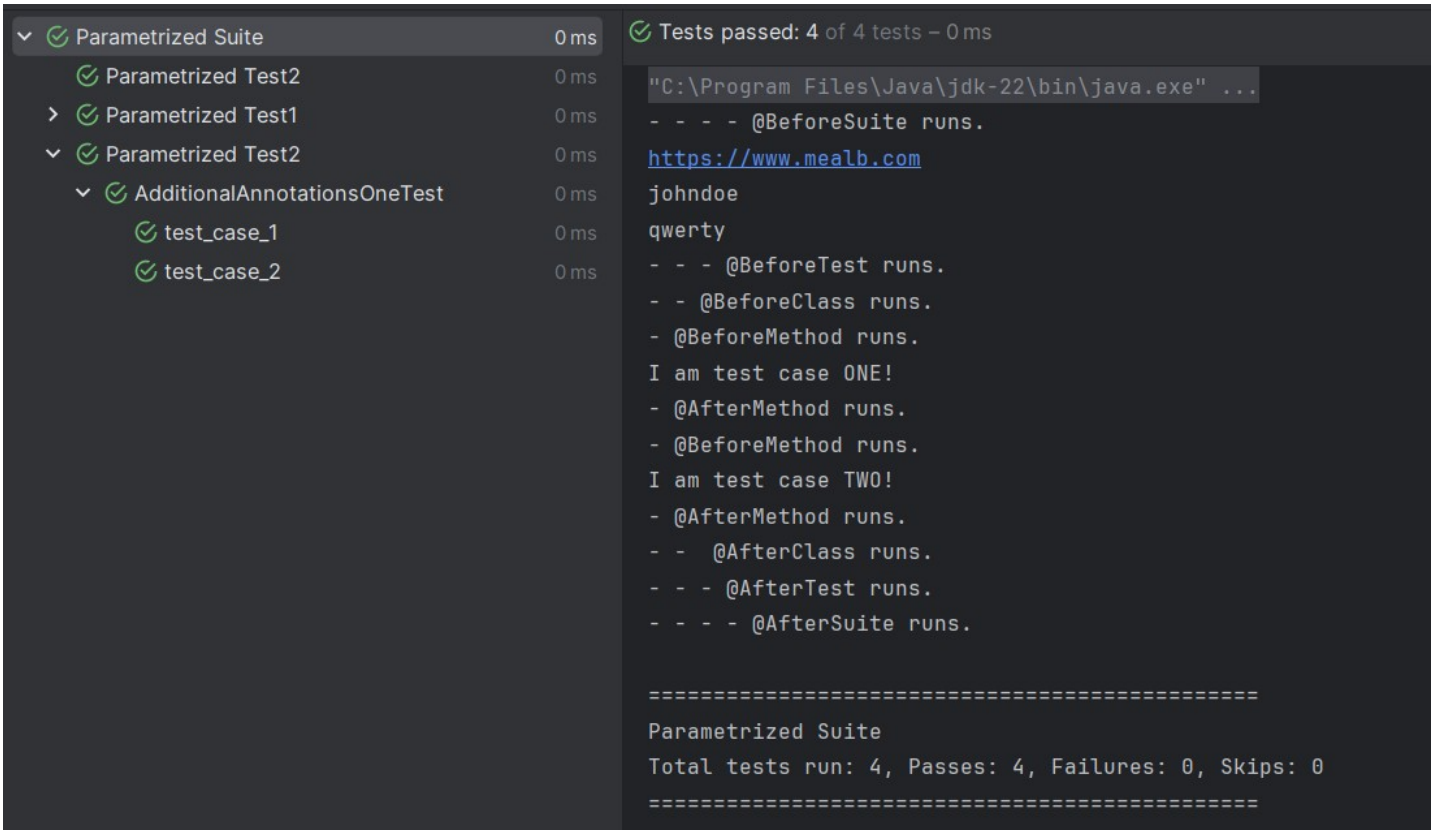
ParametersPracticeTest class

```
package testNGBasics;

import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

/*
 *In order to link parameters from xml we will utilize the @Parameters annotation
 * In the parameters annotation we will specify the name of the parameter we need the value of
 * Then we have to create a parameter in our test case so the value will be stored there
 * If you have more than one parameter, remember the order will matter
 **/
public class ParametersPracticeTest {
	@Parameters("param1")
	@Test
	public void testParametersFromXml(String parameter1) {
		System.out.println(parameter1);
	}
```

```java
	@Parameters({ "username", "password" })
	@Test
	public void testParametersFromXml2(String user, String pass) {
		System.out.println(user);
		System.out.println(pass);
	}
}
```

parameterPractice.xml file
we need to launch xml not the java class !

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Parametrized Suite">
	<!--a parameter will be declared at suite level or test level-->
	<!--each parameter will have a name (kind of key) and a value-->
	<!--a parameter declared at suite level will be accessible for all test in
that suite-->
	<!-- A parameter declared at the suite level -->
	<parameter name="param1" value="https://www.mealb.com" />

	<!-- First test -->
	<test name="Parametrized Test1">
		<!--a parameter declared in a test, will be visible only in that test
(similar to scopes in java)-->

		<!-- Parameters specific to this test -->
		<parameter name="username" value="johndoe" />
		<parameter name="password" value="qwerty" />
		<classes>
			<class name="testNGBasics.ParametersPracticeTest" />
		</classes>
	</test>

	<!-- Second test -->
	<test name="Parametrized Test2">
		<classes>
			<class name="testNGBasics.AdditionalAnnotationsOneTest" />
		</classes>
	</test>

</suite>
```

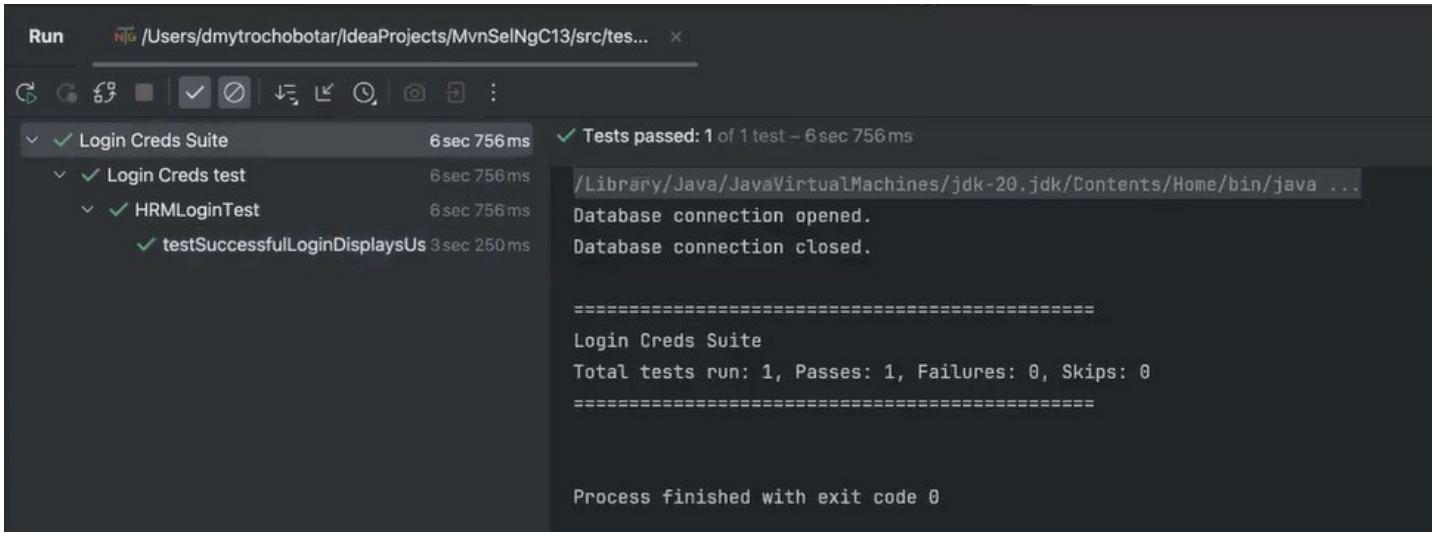


Task

Passing username and password parameters from the xml file java class:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Login Creds Suite">

    <test name="Login Creds test">

        <parameter name="username" value="yoll-student"/>
        <parameter name="password" value="Bootcamp5#"/>

        <classes>
            <class name="practice.HRMLoginTest">
                <methods>
                    <include name="testSuccessfulLoginDisplaysUsername"/>
                </methods>
            </class>
        </classes>

    </test>
</suite>
```

```java
@AfterMethod
public void tearDown() { driver.quit(); }

@BeforeClass
public void dbSetUpConnection() { System.out.println("Database connection opened."); }

@AfterClass
public void dbCleanUp() { System.out.println("Database connection closed."); }

@Parameters({"username", "password"})
@Test(priority = 0)
public void testSuccessfulLoginDisplaysUsername(String username, String password) throws I
    login(username, password);
    WebElement usernameMessage = driver.findElement(By.id("welcome"));
    String expectedUsername = "Welcome Yoll";
    String actualUsername = usernameMessage.getText();
    Assert.assertEquals(actualUsername,expectedUsername, message: "Username verification fa
}
```


Run
/Users/dmytrochobotar/IdeaProjects/MvnSelNgC13/src/tes...
Login Creds Suite 6 sec 756 ms
Login Creds test 6 sec 756 ms
HRMLoginTest 6 sec 756 ms
testSuccessfulLoginDisplaysUs 3 sec 250 ms
Tests passed: 1 of 1 test – 6 sec 756 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Database connection opened.
Database connection closed.
==========
Login Creds Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
==========
Process finished with exit code 0

Task

```
<parameter name="browserType" value="chrome"></parameter>
```

> **This specifies a parameter that will be passed to the test method. Here, the `browserType` parameter is set to "chrome," indicating that the tests in this group will be executed on the Chrome browser.**

crossBrowserTesr xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Cross Browser Tests">
    <test name="Chrome Tests">
        <parameter name="browserType" value="chrome"></parameter>
        <classes>
            <class name="testNGBasics.BrowserTypeSelectionPractice"/> </classes>
    </test>
</suite>
```

BrowserTypeSelectionPractice class

```
package testNGBasics;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.safari.SafariDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
```

```java
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class BrowserTypeSelectionPractice {
    private WebDriver driver;

    @Parameters("browserType")
    @BeforeMethod
    public void setup(String browserType) {
        //driver = new ChromeDriver();
// First way:it is used since java 12 version
        switch (browserType) {
            case "firefox" -> driver = new FirefoxDriver();
            case "safari" -> driver = new SafariDriver();
            default -> driver = new ChromeDriver();}

/* Second way:
            switch (browserType) {
                case "chrome":
                    driver = new ChromeDriver(); break;
                case "firefox":
                    driver = new FirefoxDriver(); break;
                case "safari":
                    driver = new SafariDriver(); break;
                default:
                    driver = new ChromeDriver();
    */
        driver.get("https://www.amazon.com/");

    }
    @AfterMethod
    public void teardown() {
        driver.quit();
    }
    @Test
    public void testLogin() throws InterruptedException {
        System.out.println("Testing login functionality");
        Thread.sleep( 2000);
    }

}
```

switch (browserType) explanation:

The code snippet you've provided uses the **enhanced switch statement** available in Java (introduced in Java 12 and fully integrated in Java 14). This version of the switch statement is more concise and can return values directly. Here's what the code does:

Code Breakdown

```java
switch (browserType) {

    case "firefox" -> driver = new FirefoxDriver();

    case "safari" -> driver = new SafariDriver();
```

```
    default -> driver = new ChromeDriver();

}
```

Explanation:

- **switch (browserType):**
  - The switch statement is evaluating the value of the browserType variable. This variable is expected to contain the name of the browser in the form of a string, like "firefox", "safari", or "chrome".
- **case "firefox" -> driver = new FirefoxDriver();:**
  - If browserType equals "firefox", then the FirefoxDriver is instantiated and assigned to the driver variable.

This means that the tests will be run using the Firefox browser.

And so on with chrome and safari.

default -> driver = new ChromeDriver();:

- If browserType does not match "firefox" or "safari", the default case is executed.
- Here, the ChromeDriver is instantiated and assigned to the driver variable.
- This acts as a fallback, so if no specific browser is requested, the tests will be run using Chrome by default.

- Various functionalities offered by TestNG can be found

https://testng.org/doc/documentation-main.html

https://testng.org/#_testng_documentation

Learning TestNG

Task

There is a message from Intellij :
java.lang.NullPointerException: Cannot invoke "org.openqa.selenium.WebDriver.findElement(org.openqa.selenium.By)" because "this.driver" is null

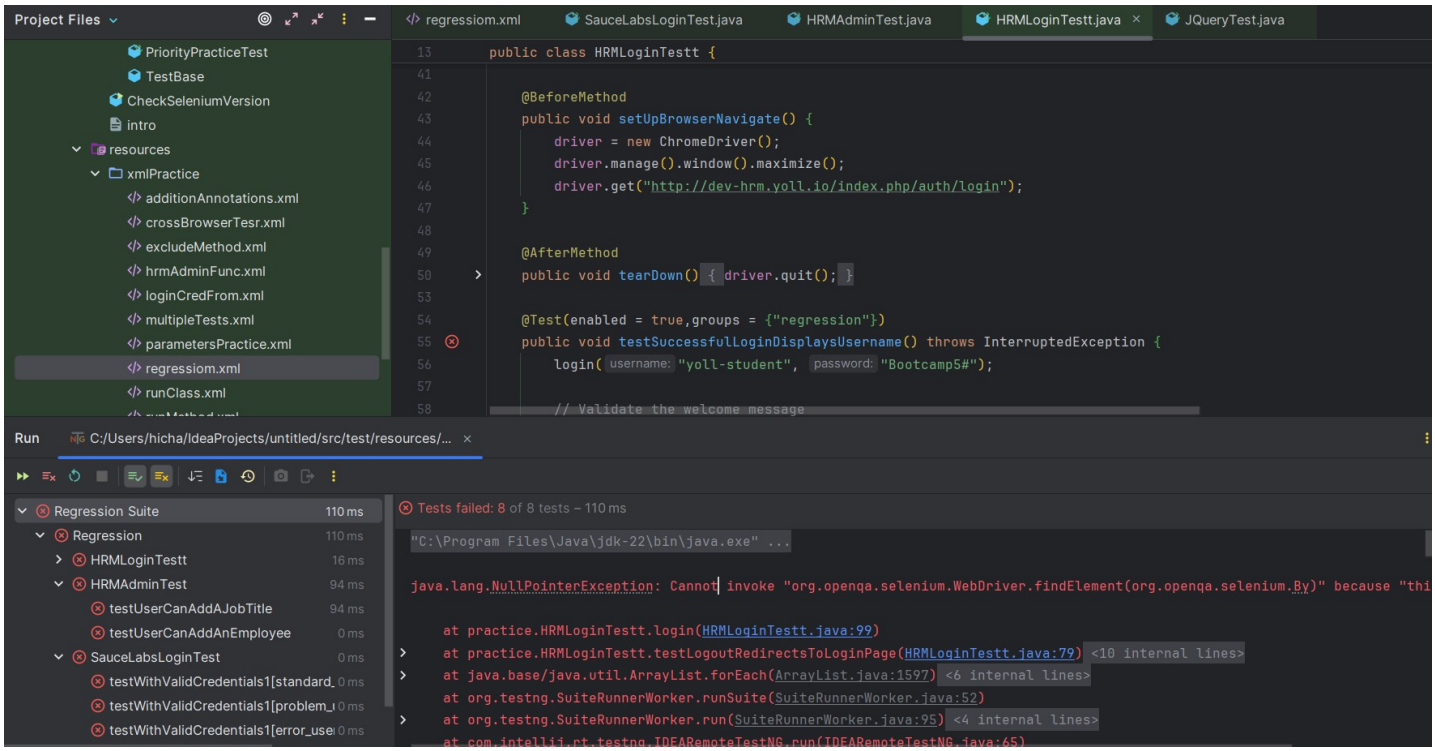
Cause:

Problem Recap:

The error occurs because the driver variable is null when you attempt to call findElement. This usually means that the WebDriver instance (driver) hasn't been initialized before you try to use it.

Steps to Fix the Issue:

1. Ensure WebDriver is Initialized: The first step is to make sure that the WebDriver (driver) is properly initialized before any operations like driver.get() or driver.findElement() are called.
2. Use Setup and Teardown Methods: If you're using TestNG, it's a good practice to initialize WebDriver in a setup method annotated with @BeforeMethod or @BeforeClass, and quit the driver in a method annotated with @AfterMethod or @AfterClass.



Solution:

we must put on each class used by the xml file:

```
@BeforeMethod(alwaysRun = true)
@AfterMethod(alwaysRun = true)
```

Explanation:

1. **@BeforeMethod(alwaysRun = true) (setUp):**

- This method is annotated with @BeforeMethod(alwaysRun = true), ensuring that it runs before every test method in the class, regardless of the groups that the test methods belong to.
- This is where the WebDriver (driver) is initialized (driver = new ChromeDriver();).

2. **Test Methods (verifyLogoIsDisplayed and verifyAddress)**:
   - These are the test methods where the WebDriver is used to navigate to Amazon's website, find elements, and make assertions.
   - Since driver is initialized in the setUp method, it will not be null when these tests run.
3. **@AfterMethod(alwaysRun = true) (tearDown)**:
   - This method ensures that the WebDriver quits after each test, closing the browser and releasing resources. The alwaysRun = true attribute ensures this cleanup happens even if the test belongs to a specific group or if the test fails.

Task:

Using **(alwaysRun = true) with @AfterMethod and @BeforeMethod for each class to avoid getting this error message: java.lang.NullPointerException: Cannot invoke "org.openqa.selenium.WebDriver.findElement(org.openqa.selenium.By)" because "this.driver" is null**

```
<classes>
     <class name="practice.HRMLoginTestt" />
     <class name="practice.HRMAdminTest" />
     <class name="practice.JQueryTest" />
     <class name="practice.SauceLabsLoginTest" />
</classes>
```

regression.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Regression Suite">
      <groups>
            <run>
                  <include name="regression" />
            </run>
      </groups>
      <test name="Regression">
            <classes>
                  <class name="practice.HRMLoginTestt" />
                  <class name="practice.HRMAdminTest" />
                  <class name="practice.JQueryTest" />
                  <class name="practice.SauceLabsLoginTest" />

                  <!--             <class name="practice.AmazonTest"/>
<class name="testNGBasics.GroupingPracticeOneTest"/>
            <class name="testNGBasics.GroupingPracticeTwoTest"/>-->
            </classes>
```

```
      </test>
</suite>
```

Classes :

```
 @BeforeMethod(alwaysRun = true)
    public void setUpBrowserNavigate() {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }

    @AfterMethod(alwaysRun = true)
    public void tearDown() {
        driver.quit();
    }


/*
In small frameworks we can have one hrmAdminFunc.xml config file.
In large frameworks where you have lots of test cases it is common that you create
multiple xml files like:
- regression.xml
- smoke.xml
Option Command + L
Control + Alt + L
 */
```

HRMAdminTest class

```
package practice;

import com.github.javafaker.Faker;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.util.List;

public class HRMAdminTest {
    private WebDriver driver;

    @BeforeMethod(alwaysRun = true)
    public void setUp() {
        driver = new ChromeDriver();
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
        driver.manage().window().maximize(); // Maximizing the window for better
interaction
    }

    @AfterMethod(alwaysRun = true)
    public void tearDown() {
        driver.quit();
```

```
    }

    /*
    Scenario: User should be able to add Job Title
    Given User is on homepage
    User clicks on Admin > Job > Job Titles
    User can see Add and Delete buttons are displayed
    User can add a new Job Title
    Then newly Added Job Information is displayed in Table
    */
    @Test(groups = {"regression"})
    public void testUserCanAddAJobTitle() throws InterruptedException {
        Faker mockData = new Faker();

        /* Initialize data to use */
        String username = "yoll-student";
        String password = "Bootcamp5#";
        String jobTitleText = mockData.job().title();
        String jobTitleDescription = mockData.chuckNorris().fact();

        /* Log in */
        login(username, password);

        /* Locate the web elements and Interact with web elements */

        // Navigate to Admin > Job > Job Titles
        WebElement adminTab = driver.findElement(By.id("menu_admin_viewAdminModule"));
        adminTab.click();
        Thread.sleep(1000);

        WebElement jobDropdown = driver.findElement(By.id("menu_admin_Job"));
        jobDropdown.click();
        Thread.sleep(1000);

        WebElement jobTitleOption =
driver.findElement(By.id("menu_admin_viewJobTitleList"));
        jobTitleOption.click();
        Thread.sleep(1000);

        // Validate Add and Delete buttons are displayed
        WebElement addButton = driver.findElement(By.id("btnAdd"));
        WebElement deleteButton = driver.findElement(By.id("btnDelete"));

        Assert.assertTrue(addButton.isDisplayed(), "Add Button Verification Failed.");
        Assert.assertTrue(deleteButton.isDisplayed(), "Delete Button Verification
Failed.");

        // Add new job title
        addButton.click();
        WebElement jobTitleInputBox = driver.findElement(By.id("jobTitle_jobTitle"));
        WebElement jobDescriptionInputBox =
driver.findElement(By.id("jobTitle_jobDescription"));

        jobTitleInputBox.sendKeys(jobTitleText);
        Thread.sleep(2000);
```

```
        jobDescriptionInputBox.sendKeys(jobTitleDescription);
        Thread.sleep(2000);

        // Save the job title
        WebElement saveButton = driver.findElement(By.id("btnSave"));
        saveButton.click();
        Thread.sleep(2000);

        // Verify the newly added job title appears in the table
        List<WebElement> jobTitles =
driver.findElements(By.xpath("//table/tbody/tr/td[2]"));
        boolean jobTitleSuccessfullyAdded = false;

        for (WebElement jobTitle : jobTitles) {
            if (jobTitle.getText().equals(jobTitleText)) {
                jobTitleSuccessfullyAdded = true;
                break;
            }
        }

        Assert.assertTrue(jobTitleSuccessfullyAdded, "Job title was not successfully
added.");
    }
    @Test(groups = {"regression"})
    public void testUserCanAddAnEmployee() throws InterruptedException {
        /* Login to HRM */
        String username = "yoll-student";
        String password = "Bootcamp5#";
        login(username, password);
        /* Navigate to Add Employee Page */
        WebElement pimTab = driver.findElement(By.id("menu_pim_viewPimModule"));
        pimTab.click();
        Thread.sleep(1000);
        WebElement addButton = driver.findElement(By.id("btnAdd"));
        addButton.click();
        Thread.sleep(1000);
        WebElement firstNameInputBox = driver.findElement(By.id("firstName"));
        WebElement middleNameInputBox = driver.findElement(By.id("middleName"));
        WebElement lastNameInputBox = driver.findElement(By.id("lastName"));

        Faker faker = new Faker();
        String firstName = faker.name().firstName();
        String middleName = faker.funnyName().name();
        String LastName = faker.name().lastName();

        firstNameInputBox.sendKeys(firstName);
        Thread.sleep(1000);
        middleNameInputBox.sendKeys(middleName);
        Thread.sleep(1000);
        lastNameInputBox.sendKeys(LastName);
        Thread.sleep(1000);


        WebElement idField = driver.findElement(By.id("employeeId"));
```

```
        /* Option 1 */
        String fillingId = idField.getAttribute("value");
        boolean isNotEmpty = !fillingId.isEmpty();
        Assert.assertTrue(isNotEmpty, "Id field verification failed.");
        /* Option 2 */
        // Assert.assertFalse (idField.getAttribute("value").isEmpty(), "Id field
verification failed.");
        WebElement saveButton = driver.findElement(By.id("btnSave"));
        saveButton.click();
        /* Extract the personal information after the employee was created */
        WebElement actualFirstName =
driver.findElement(By.id("personal_txtEmpFirstName"));
        WebElement actualMiddleName =
driver.findElement(By.id("personal_txtEmpMiddleName"));
        WebElement actualLastName = driver.findElement(By.id("personal_txtEmpLastName"));
        WebElement actualId = driver.findElement(By.id("personal_txtEmployeeId"));


        /* Assert personal Details */
        Assert.assertEquals(actualFirstName.getAttribute("value"),firstName,
                "First name verification failed");
        Assert.assertEquals(actualMiddleName.getAttribute("value"),middleName, "Middle
name verification failed");
        Assert.assertEquals(actualLastName.getAttribute("value"),LastName, "Last name
verification failed");

        Assert.assertEquals(actualId.getAttribute("value"),fillingId, "Id verification
failed");
    }

    // Method to perform login
    public void login(String username, String password) throws InterruptedException {
        WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
        WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
        WebElement loginButton = driver.findElement(By.id("btnLogin"));

        usernameInputBox.sendKeys(username);
        passwordInputBox.sendKeys(password);
        loginButton.click();
        Thread.sleep(2000);
    }
}
```

SauceLabsLoginTest class:

```
package practice;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.DataProvider;
```

```java
import org.testng.annotations.Test;

public class SauceLabsLoginTest {
    private WebDriver driver;

    /* User should be able to successfully login */
    @BeforeMethod(alwaysRun = true)
    public void setUpBrowserNavigate() {

        driver = new ChromeDriver();
// driver.manage().window().maximize();
        driver.get("https://www.saucedemo.com/");
    }

    @AfterMethod(alwaysRun = true)
    public void tearDown() {
        driver.quit();
    }

    @Test(dataProvider ="sauceLabsLoginCredentials",groups = {"regression"})
    public void testWithValidCredentials1(String username,String password ) throws
InterruptedException {

        WebElement usernameInputBox = driver.findElement(By.id("user-name"));
        WebElement passwordInputBox = driver.findElement(By.id("password"));
        WebElement LoginButton = driver.findElement(By.id("login-button"));
        usernameInputBox.sendKeys(username);
        passwordInputBox.sendKeys(password);
        LoginButton.click();
        Thread.sleep(1000);
// How to understand if an element is present on the page
// A: Use findElements() and check the size of the list
        boolean isProductLabelPresent = driver.findElements
                (By.xpath("//*[text() = 'Products']")).size() == 1;
        Assert.assertTrue(isProductLabelPresent, "Login verification failed!");
    }


    // Data Providers
    @DataProvider(name = "sauceLabsLoginCredentials")
    public Object[][] dataSet() {
        Object[][] dataSet = {
                {"standard_user", "secret_sauce"},
                {"problem_user", "secret_sauce"},
                {"error_user", "secret_sauce"},
                {"visual_user", "secret_sauce"}
        };
        return dataSet;


    }}
```

HRMLoginTestt class:

```java
package practice;

import com.github.javafaker.Faker;
import jdk.jfr.Enabled;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.*;

public class HRMLoginTestt {

    private WebDriver driver;

    /*
    If we want to disable a test case not execute
    *- comment it out
    * - remove it
    * - TestNG provides you a property that can disable a method
    * - * To disable a method we mark it as enabled = false
        By default all test are enabled = true
    **/

    @BeforeClass
    public void dbSetUpConnection() {
        System.out.println("Database connection opened.");
    }

    /*
        Summary
        JUnit: @BeforeClass and @AfterClass must be static
        TestNG: @BeforeClass and @AfterClass do not need to be static.
        BE CAREFUL WHEN CHOOSING THE ANNOTATIONS, IT MUST BE FOR TestNG NOT JUnit !
         */
    @AfterClass
    public void dbCleanUp() {
        System.out.println("Database connection closed.");
    }

    @BeforeMethod(alwaysRun = true)
    public void setUpBrowserNavigate() {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }

    @AfterMethod(alwaysRun = true)
    public void tearDown() {
        driver.quit();
    }
```

```
    @Test(enabled = true,groups = {"regression"})
    public void testSuccessfulLoginDisplaysUsername() throws InterruptedException {
        login("yoll-student", "Bootcamp5#");

        // Validate the welcome message
        WebElement usernameMessage = driver.findElement(By.id("welcome"));
        String expectedUsername = "Welcome Yoll";
        String actualUsername = usernameMessage.getText();
        Assert.assertEquals(expectedUsername, actualUsername, "Username verification
failed.");
    }


    @Test(dataProviderClass = TestData.class,
            dataProvider = "hrmLoginDataSet",
            groups = {"regerssion"})
    public void testLoginWithInvalidCredentialsDisplaysErrorMessage(String username,
String password, String expectedErrorMessage) throws InterruptedException {
        // username and password are provided from TestData Class that contains 2D Array
Object
        login(username, password);
        WebElement errorMessageElement = driver.findElement(By.id("spanMessage"));
        String actualErrorMessage = errorMessageElement.getText();
        Assert.assertEquals(actualErrorMessage, expectedErrorMessage, "Error message
verification failed.");
    }

    @Test(groups = {"regression"})
    public void testLogoutRedirectsToLoginPage() throws InterruptedException {
        login("yoll-student", "Bootcamp5#");

        // Locate and click on the dropdown menu
        WebElement dropdownMenu = driver.findElement(By.id("welcome"));
        dropdownMenu.click();

        // Wait for the logout option to be available and click it
        Thread.sleep(5000);
        WebElement logoutOption = driver.findElement(By.xpath("//a[text()='Logout']"));
        logoutOption.click();

        // Wait for the logout to complete
        Thread.sleep(2000);

        // Validate if the login page is displayed
        WebElement loginHeaderPanel = driver.findElement(By.id("logInPanelHeading"));
        Assert.assertTrue(loginHeaderPanel.isDisplayed(), "Login panel verification
failed.");
    }

    public void login(String username, String password) throws InterruptedException {
        WebElement usernameInputBox = driver.findElement(By.id("txtUsername"));
        WebElement passwordInputBox = driver.findElement(By.id("txtPassword"));
        WebElement loginButton = driver.findElement(By.id("btnLogin"));
```

```
        usernameInputBox.sendKeys(username);
        Thread.sleep(2000);
        passwordInputBox.sendKeys(password);
        Thread.sleep(2000);
        loginButton.click();
        Thread.sleep(2000);
    }
    /*
Scenario: As an admin I should be able to successfully add Employee
Given I'm logged in to HRM with Admin Account
And I click on PIM Add Employee
Then I fill out First Name, Middle Name, Last Name
And I verify Employee number is not empty
Then I click on Save
And I verify new account got created with correct details
*/

    @Test
    public void testUserCanAddAnEmployee() throws InterruptedException {
        /* Login to HRM */
        String username = "yoll-student";
        String password = "Bootcamp5#";
        login(username, password);
        /* Navigate to Add Employee Page */
        WebElement pimTab = driver.findElement(By.id("menu_pim_viewPimModule"));
        pimTab.click();
        Thread.sleep(1000);
        WebElement addButton = driver.findElement(By.id("btnAdd"));
        addButton.click();
        Thread.sleep(1000);
        WebElement firstNameInputBox = driver.findElement(By.id("firstName"));
        WebElement middleNameInputBox = driver.findElement(By.id("middleName"));
        WebElement lastNameInputBox = driver.findElement(By.id("lastName"));

        Faker faker = new Faker();
        String firstName = faker.name().firstName();
        String middleName = faker.funnyName().name();
        String LastName = faker.name().lastName();

        firstNameInputBox.sendKeys(firstName);
        Thread.sleep(1000);
        middleNameInputBox.sendKeys(middleName);
        Thread.sleep(1000);
        lastNameInputBox.sendKeys(LastName);
        Thread.sleep(1000);


        WebElement idField = driver.findElement(By.id("employeeId"));

        /* Option 1 */
        String fillingId = idField.getAttribute("value");
        boolean isNotEmpty = !fillingId.isEmpty();
        Assert.assertTrue(isNotEmpty, "Id field verification failed.");
        /* Option 2 */
```

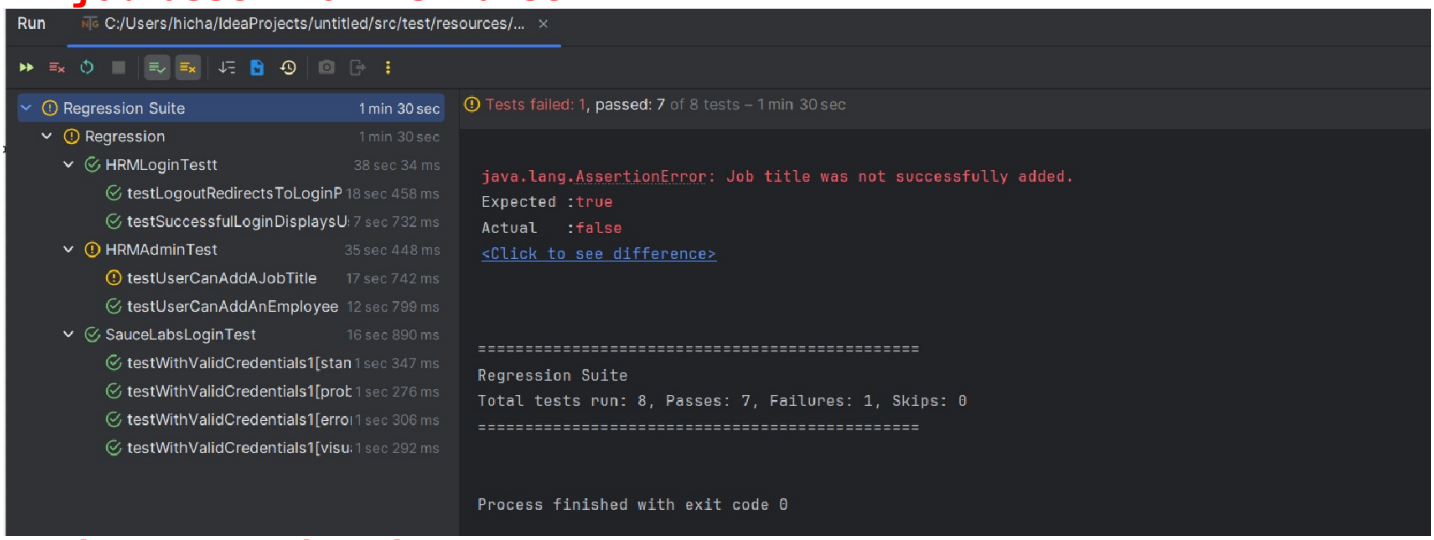
```
        // Assert.assertFalse (idField.getAttribute("value").isEmpty(), "Id field
verification failed.");
        WebElement saveButton = driver.findElement(By.id("btnSave"));
        saveButton.click();
        /* Extract the personal information after the employee was created */
        WebElement actualFirstName =
driver.findElement(By.id("personal_txtEmpFirstName"));
        WebElement actualMiddleName =
driver.findElement(By.id("personal_txtEmpMiddleName"));
        WebElement actualLastName = driver.findElement(By.id("personal_txtEmpLastName"));
        WebElement actualId = driver.findElement(By.id("personal_txtEmployeeId"));


        /* Assert personal Details */
        Assert.assertEquals(actualFirstName.getAttribute("value"),firstName,
                "First name verification failed");
        Assert.assertEquals(actualMiddleName.getAttribute("value"),middleName, "Middle
name verification failed");
        Assert.assertEquals(actualLastName.getAttribute("value"),LastName, "Last name
verification failed");

        Assert.assertEquals(actualId.getAttribute("value"),fillingId, "Id verification
failed");
    }
}
```

JqueryTest class:

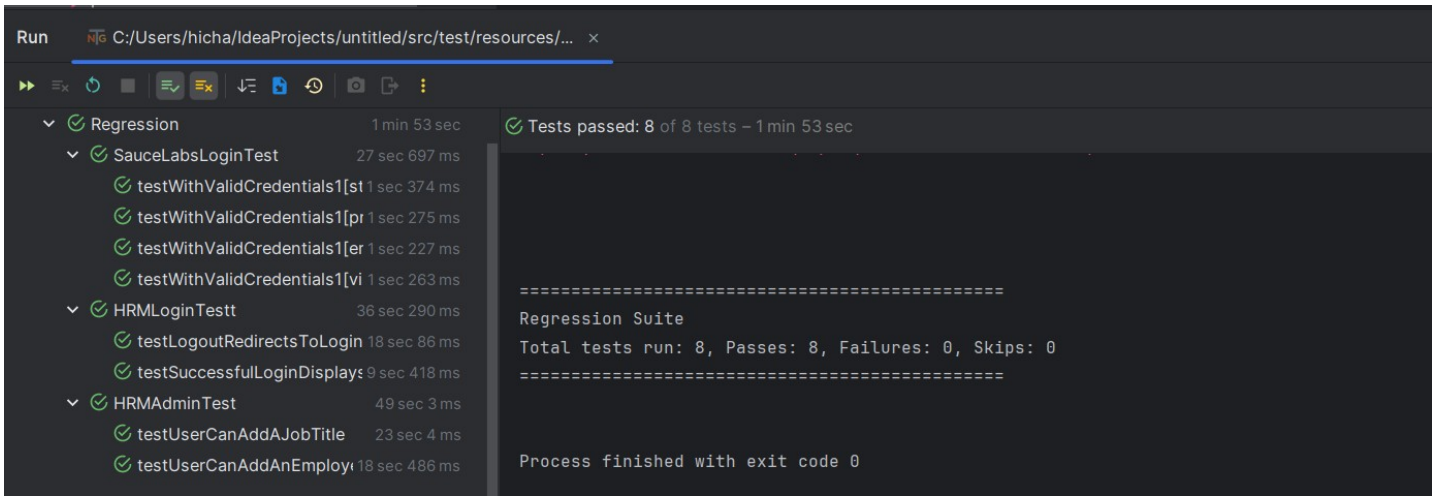
```
package practice;

import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class JQueryTest {

     /*   Go to https://demos.jquerymobile.com/1.4.3/forms-disabled/
        Assert that textArea field is disabled
*
        * Given user navigates to "https://demos.jquerymobile.com/1.4.3/forms-disabled/"
                *When user lands on Form Disabled Page
* Then textArea field is disabled
**/
        @Test
        public void verifyTextAreaDisabled() {
            WebDriver driver = new ChromeDriver();
            driver.get("https://demos.jquerymobile.com/1.4.3/forms-disabled/");
            WebElement textArea = driver.findElement(By.id("textarea-1"));
```

```
            textArea.isEnabled();
// If a tag is disabled it will have attribute disabled
// In selenium we can only identify if a tag is Enabled.
// isEnabled() will give us back true if we can interact with the element
// and false if we cannot
// The test is about we verify that we cannot interact with element
            // The test is about we verify that we cannot interact with element
            Assert.assertFalse("The text area is not disabled", textArea.isEnabled());
            // The test is about we verify that we cannot interact with element
            Assert.assertFalse("The text area is not disabled", textArea.isEnabled());
            driver.quit();

            driver.quit();
        }
}
```

If job assertion is false


Run
C:/Users/hicha/IdeaProjects/untitled/src/test/resources/...
Regression Suite 1 min 30 sec
Regression 1 min 30 sec
HRMLoginTestt 38 sec 34 ms
testLogoutRedirectsToLoginP 18 sec 458 ms
testSuccessfulLoginDisplaysU 7 sec 732 ms
HRMAdminTest 35 sec 448 ms
testUserCanAddAJobTitle 17 sec 742 ms
testUserCanAddAnEmployee 12 sec 799 ms
SauceLabsLoginTest 16 sec 890 ms
testWithValidCredentials1[stan 1 sec 347 ms
testWithValidCredentials1[prob 1 sec 276 ms
testWithValidCredentials1[erro 1 sec 306 ms
testWithValidCredentials1[visu 1 sec 292 ms
Tests failed: 1, passed: 7 of 8 tests – 1 min 30 sec
java.lang.AssertionError: Job title was not successfully added.
Expected :true
Actual :false
<Click to see difference>
==========
Regression Suite
Total tests run: 8, Passes: 7, Failures: 1, Skips: 0
==========
Process finished with exit code 0

If job assertion is true


Run
C:/Users/hicha/IdeaProjects/untitled/src/test/resources/...
Regression 1 min 53 sec
SauceLabsLoginTest 27 sec 697 ms
testWithValidCredentials1[st 1 sec 374 ms
testWithValidCredentials1[pr 1 sec 275 ms
testWithValidCredentials1[er 1 sec 227 ms
testWithValidCredentials1[vi 1 sec 263 ms
HRMLoginTestt 36 sec 290 ms
testLogoutRedirectsToLogin 18 sec 86 ms
testSuccessfulLoginDisplays 9 sec 418 ms
HRMAdminTest 49 sec 3 ms
testUserCanAddAJobTitle 23 sec 4 ms
testUserCanAddAnEmploy 18 sec 486 ms
Tests passed: 8 of 8 tests – 1 min 53 sec
==========
Regression Suite
Total tests run: 8, Passes: 8, Failures: 0, Skips: 0
==========
Process finished with exit code 0

Explanation of the List WebElement and the loop for the existing job Assertion:

Purpose of the Code:

The test automates the process of adding a new job title to an application. It navigates through the UI, interacts with elements like buttons and input fields, adds a new job title, and verifies if the newly added job title appears in a table.

The Key Section: `List<WebElement> jobTitles`

In this section, the test collects all job titles displayed in a table on the webpage and then checks if the newly added job title is present in that list.

```
List<WebElement> jobTitles = driver.findElements(By.xpath("//table/tbody/tr/td[2]"));
boolean jobTitleSuccessfullyAdded = false;

for (WebElement jobTitle : jobTitles) {
    if (jobTitle.getText().equals(jobTitleText)) {
        jobTitleSuccessfullyAdded = true;
        break;
    }
}

Assert.assertTrue(jobTitleSuccessfullyAdded, "Job title was not successfully added.");
```

Explanation of `List<WebElement>`:

1. **`driver.findElements(By.xpath("//table/tbody/tr/td[2]"))`**:
    - This line uses Selenium's `findElements()` method to locate multiple elements on the page.
    - The XPath `//table/tbody/tr/td[2]` targets all the `<td>` elements in the second column of every row (`<tr>`) inside the table's `<tbody>`. The second column typically contains job titles.
    - The result is a **list of `WebElement` objects**, each representing a job title in the table.
2. **Iterating through the List**:
    - The code then iterates over the list of `WebElement` objects using a `for` loop:

      ```
      for (WebElement jobTitle : jobTitles) {
      ```

        - Each `WebElement` in the `jobTitles` list corresponds to one of the job titles displayed in the second column of the table.
        - The method `jobTitle.getText()` extracts the visible text of each job title (i.e., the job title string).
3. **Checking for the New Job Title**:

- Inside the loop, the code compares the text of each job title with the `jobTitleText` (which is the job title that was just added):

  ```
  if (jobTitle.getText().equals(jobTitleText)) {
  ```

- If the job title matches, the flag `jobTitleSuccessfullyAdded` is set to `true` and the loop is exited.

4. **Assertion**:
   - After iterating through the list, the test asserts that the new job title was found in the table:

     ```
     Assert.assertTrue(jobTitleSuccessfullyAdded, "Job title was not successfully added.");
     ```

   - If the job title was not found, the test will fail, indicating that the new job title did not appear in the list.

Summary of `List<WebElement>` Usage:

- **Purpose**: The `List<WebElement>` is used to store all the job titles present in a specific column of a table.
- **Role**: The loop iterates through this list to verify if the newly added job title appears in the table, ensuring that the job title was successfully added to the system.
- **Outcome**: If the new job title is found in the table, the test passes. If not, the test fails with an assertion error.

This approach is very common in Selenium tests when you need to validate that a certain piece of dynamic content (like a job title in this case) has been successfully added to a webpage.

parallel and thread-count

Parallel Execution & Cross-Browser Testing

- In TestNG, we can also use XML file to configure parallel testing.
  - We specify how many threads to run in parallel.
  - `thread-count="2"`
- We can run multiple methods, classes or tests in parallel:
  - `parallel="methods"`
  - `parallel ="classes"`
  - `parallel="tests"`

- We can also run tests in parallel using different kind of browsers: Chrome, Firefox, Edge etc.
  - We achieve it by passing a browser parameter from XML file
  - For this we use @Parameters option and pass the browser name from xml file as parameter.

```
<suite name="Regression Suite" parallel="classes" thread-count="4">
```

TestNG allows you to configure test execution using XML, enabling features like **parallel testing, grouping, and managing test runs across multiple Classes / Methodes / Tests at the same time.**

- The **suite name** (`"Regression Suite"`) indicates a suite intended for running regression tests.
- **`parallel="classes"`** means multiple test classes will run in parallel. This means that TestNG will run the test methods in different classes in parallel, rather than running them sequentially. Each test class will be executed in its own thread, allowing multiple classes to run at the same time.
- **`thread-count="4"`** defines the number of concurrent threads (4 test classes can run simultaneously).

WE CAN USE THE NAME OF THE PACKAGES INSTEAD OF THE CLASS BY SPLITTING THE CLASSES BY THE NUMBER OF THE `thread-count="4"`**:**

```
<suite name="Regression Suite" parallel="classes" thread-count="4">

<packages>
   <package name="practice"/>
   <package name="testNGBasics"/>
</packages>
         <!--
        <classes>
            <class name="practice.SauceLabsLoginTest"/>
```

```
<class name="practice.HRMLoginTestt"/>
<class name="practice.HRMAdminTest"/>
<class name="practice.JQueryTest"/>
-->
```

CROSS BROWSER : RUNNING DIFFERENT BROWSER AT THE SAME TIME

First Way:

The xml file runs BrowserTypeSelectionPractice class on Chrome, Firefox, and Safari browsers ONE AFTER ONE.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Cross Browser Tests">
	<test name="Chrome Tests">
		<parameter name="browserType" value="chrome" />
		<classes>
			<class name="testNGBasics.BrowserTypeSelectionPractice" />
		</classes>
	</test>

	<test name="Firefox Tests">
		<parameter name="browserType" value="firefox" />
		<classes>
			<class name="testNGBasics.BrowserTypeSelectionPractice" />
		</classes>
	</test>

	<test name="Safari Tests">
		<parameter name="browserType" value="safari" />
		<classes>
			<class name="testNGBasics.BrowserTypeSelectionPractice" />
		</classes>
	</test>
</suite>
```

Second Way: using PARALLEL AND THREAD COUNT

The xml file runs BrowserTypeSelectionPractice class on Chrome, Firefox, and Safari browsers AT THE SAME TIME.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
```

```xml
<suite name="Cross Browser Tests" parallel="tests" thread-count="3">
	<test name="Chrome Tests">
		<parameter name="browserType" value="chrome" />
		<classes>
			<class name="testNGBasics.BrowserTypeSelectionPractice" />
		</classes>
	</test>

	<test name="Firefox Tests">
		<parameter name="browserType" value="firefox" />
		<classes>
			<class name="testNGBasics.BrowserTypeSelectionPractice" />
		</classes>
	</test>

	<test name="Safari Tests">
		<parameter name="browserType" value="safari" />
		<classes>
			<class name="testNGBasics.BrowserTypeSelectionPractice" />
		</classes>
	</test>
</suite>
```

TESTNG HTML REPORT

- In order to get HTML report, first you need to run your test
- Then check test-output/default suite folder
- It will create an **index.html report** file under test-output folder
  - If it is not showing then Run → Edit Run Configuration → Listeners → Check Default reporters.
- There is also another report that gets generated: Emailable-report.html
- **Testng-failed.xml** → is an xml file that gets generated.
  - If you run this xml file, then only failed test cases from last run, will be executed.

Test

Tests passed/Failed/Skipped:	5/0/0
Started on:	Mon Dec 30 00:39:20 EST 2019
Total time:	0 seconds (47 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**
m1 Test class: Practice.BeforeTestAnnot.TC1		0	Practice.BeforeTestAnnot.TC1@50d0686
m2 Test class: Practice.BeforeTestAnnot.TC1		0	Practice.BeforeTestAnnot.TC1@50d0686
m3 Test class: Practice.BeforeTestAnnot.TC2		0	Practice.BeforeTestAnnot.TC2@7a3435bd

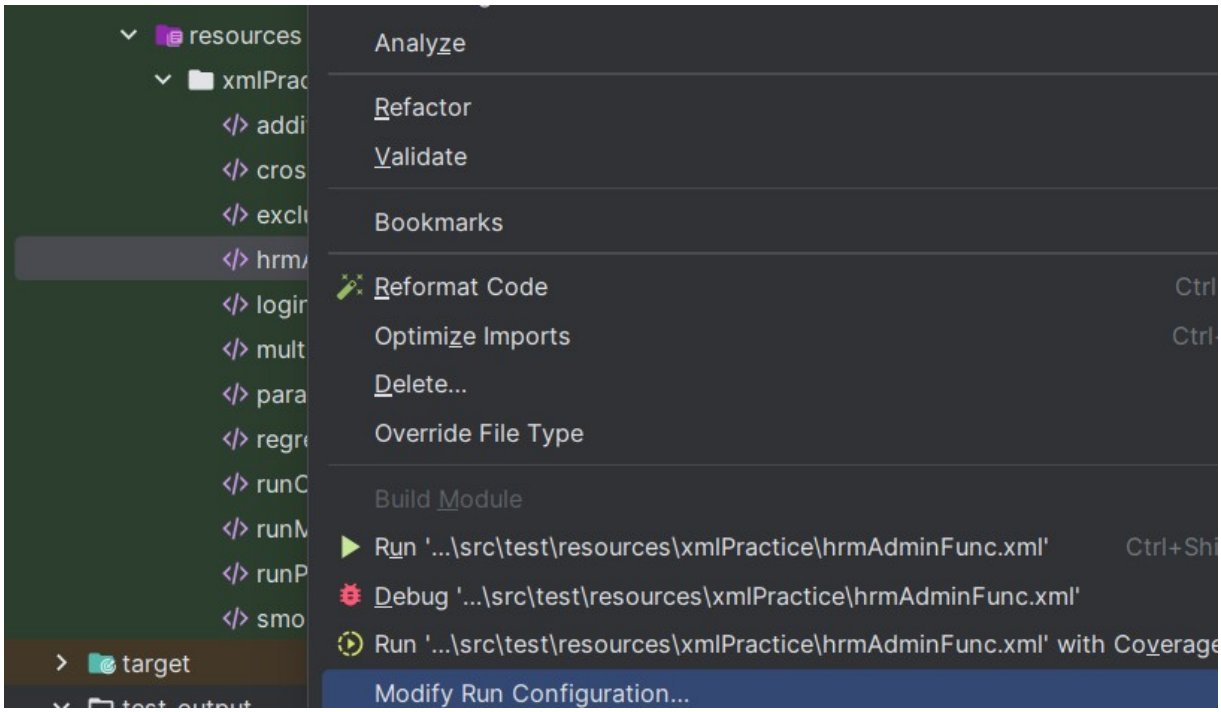
First, ***RIGHT CLICK ON XML FILE***/ choose ***MODIFY RUN CONFIGURATION***.

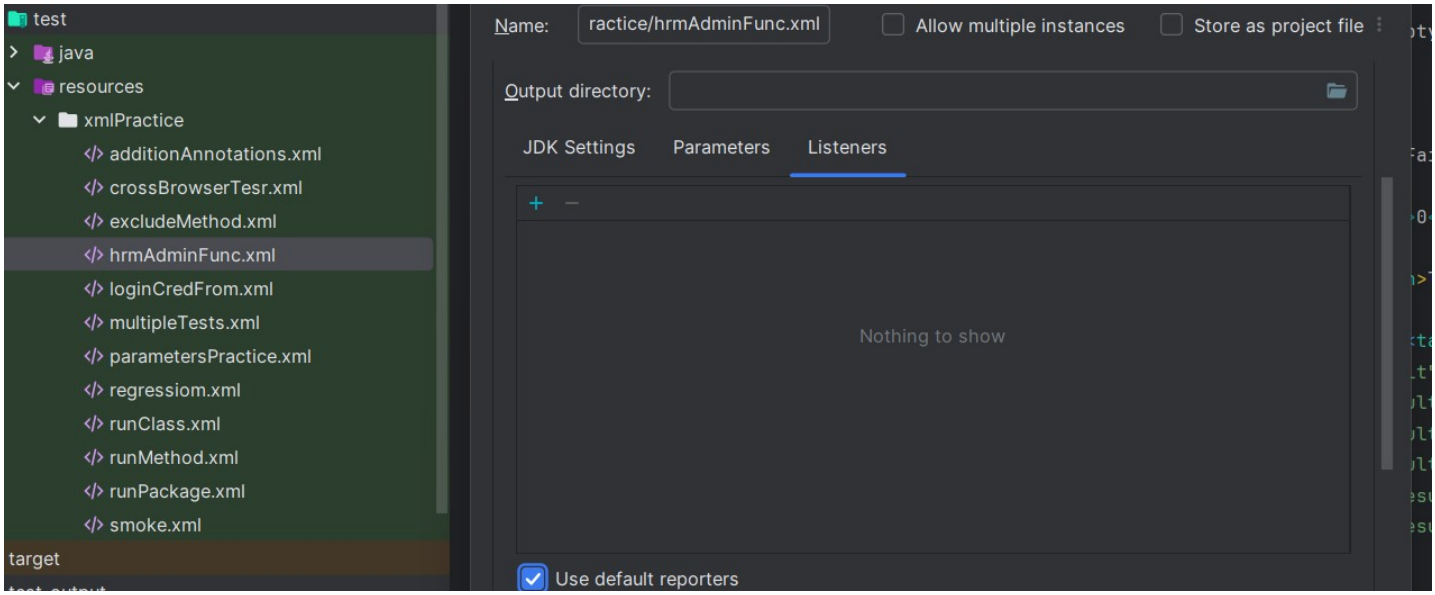
Second, GO TO ***LISTNERS***/ check the box ***: USE DEFAULTS REPORTERS***/ Apply / Ok.

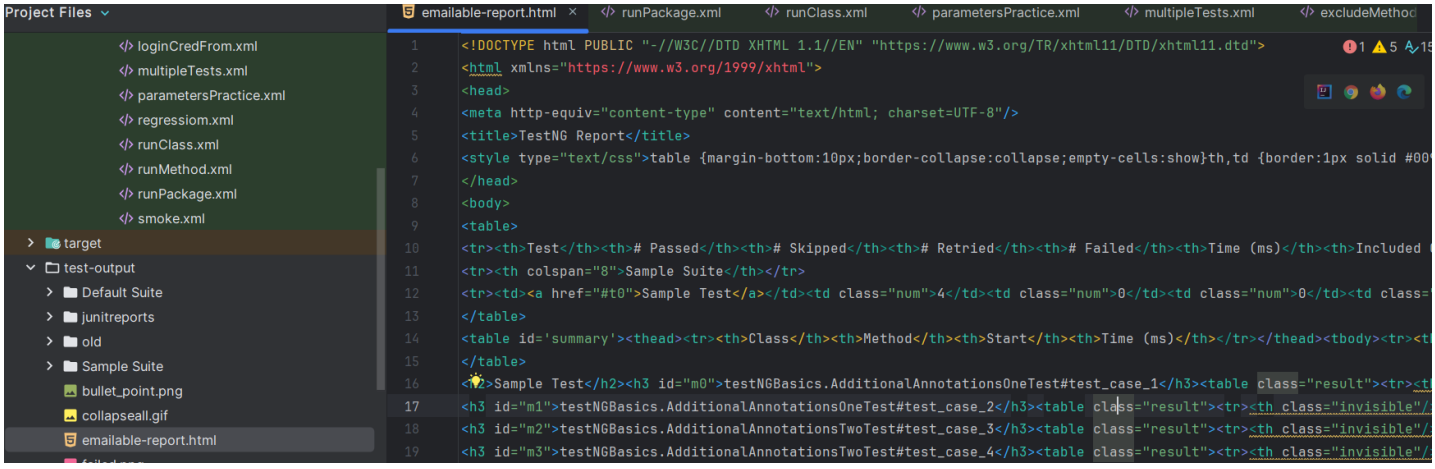
Third, Nothing will happen until you ***run the same xml file***.

You can see under **target**: **test-output** folder/ click on **emailable-report.html**.

Open it from the right side with a browser.




test
java
resources
xmlPractice
additionAnnotations.xml
crossBrowserTesr.xml
excludeMethod.xml
hrmAdminFunc.xml
loginCredFrom.xml
multipleTests.xml
parametersPractice.xml
regressiom.xml
runClass.xml
runMethod.xml
runPackage.xml
smoke.xml
target
Name:
ractice/hrmAdminFunc.xml
Allow multiple instances
Store as project file
Output directory:
JDK Settings
Parameters
Listeners
Nothing to show
Use default reporters


Project Files
loginCredFrom.xml
multipleTests.xml
parametersPractice.xml
regressiom.xml
runClass.xml
runMethod.xml
runPackage.xml
smoke.xml
target
test-output
Default Suite
junitreports
old
Sample Suite
bullet_point.png
collapseall.gif
emailable-report.html
emailable-report.html
runPackage.xml
runClass.xml
parametersPractice.xml
multipleTests.xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "https://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="https://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
<title>TestNG Report</title>
<table id='summary'><thead><tr><th>Class</th><th>Method</th><th>Start</th><th>Time (ms)</th></tr></thead><tbody><tr><t
<h3 id="m1">testNGBasics.AdditionalAnnotationsOneTest#test_case_2</h3><table class="result"><tr><th class="invisible"/
<h3 id="m2">testNGBasics.AdditionalAnnotationsTwoTest#test_case_3</h3><table class="result"><tr><th class="invisible"/
<h3 id="m3">testNGBasics.AdditionalAnnotationsTwoTest#test_case_4</h3><table class="result"><tr><th class="invisible"/

Test	# Passed	# Skipped	# Retried	# Failed	Time (ms)	Included Groups	Excluded Groups
Sample Suite							
Sample Test	4	0	0	0	47		

Class	Method	Start	Time (ms)
Sample Suite			
Sample Test — passed			
testNGBasics.AdditionalAnnotationsOneTest	test_case_1	1725723761315	0
	test_case_2	1725723761331	0
testNGBasics.AdditionalAnnotationsTwoTest	test_case_3	1725723761331	0
	test_case_4	1725723761331	0

Sample Test

testNGBasics.AdditionalAnnotationsOneTest#test_case_1

back to summary

---

testNGBasics.AdditionalAnnotationsOneTest#test_case_2

back to summary

---

testNGBasics.AdditionalAnnotationsTwoTest#test_case_3

back to summary

---

testNGBasics.AdditionalAnnotationsTwoTest#test_case_4

back to summary

Time stamps: it can be converted to know the exact time using this website: Epoch & Unix Timestamp Conversion Tools

Epoch Converter - Unix Timestamp Converter

	Start	T
1	1725677617888	
2	1725677617904	
3	1725677617920	
4	1725677617920	

EpochConverter

Epoch & Unix Timestamp Conversion Tools

The current Unix epoch time is 1725809975

Convert epoch to human-readable date and vice versa

1725809926 Timestamp to Human date [batch convert]

Supports Unix timestamps in seconds, milliseconds, microseconds and nanoseconds.

Mon	Day	Yr	Hr	Min	Sec	
9	/ 8	/ 2024	15 :	38 :	46	GMT Human date to Timestamp

To view the xml file, click on index.html under test-output folder:

Xml file used for our case :

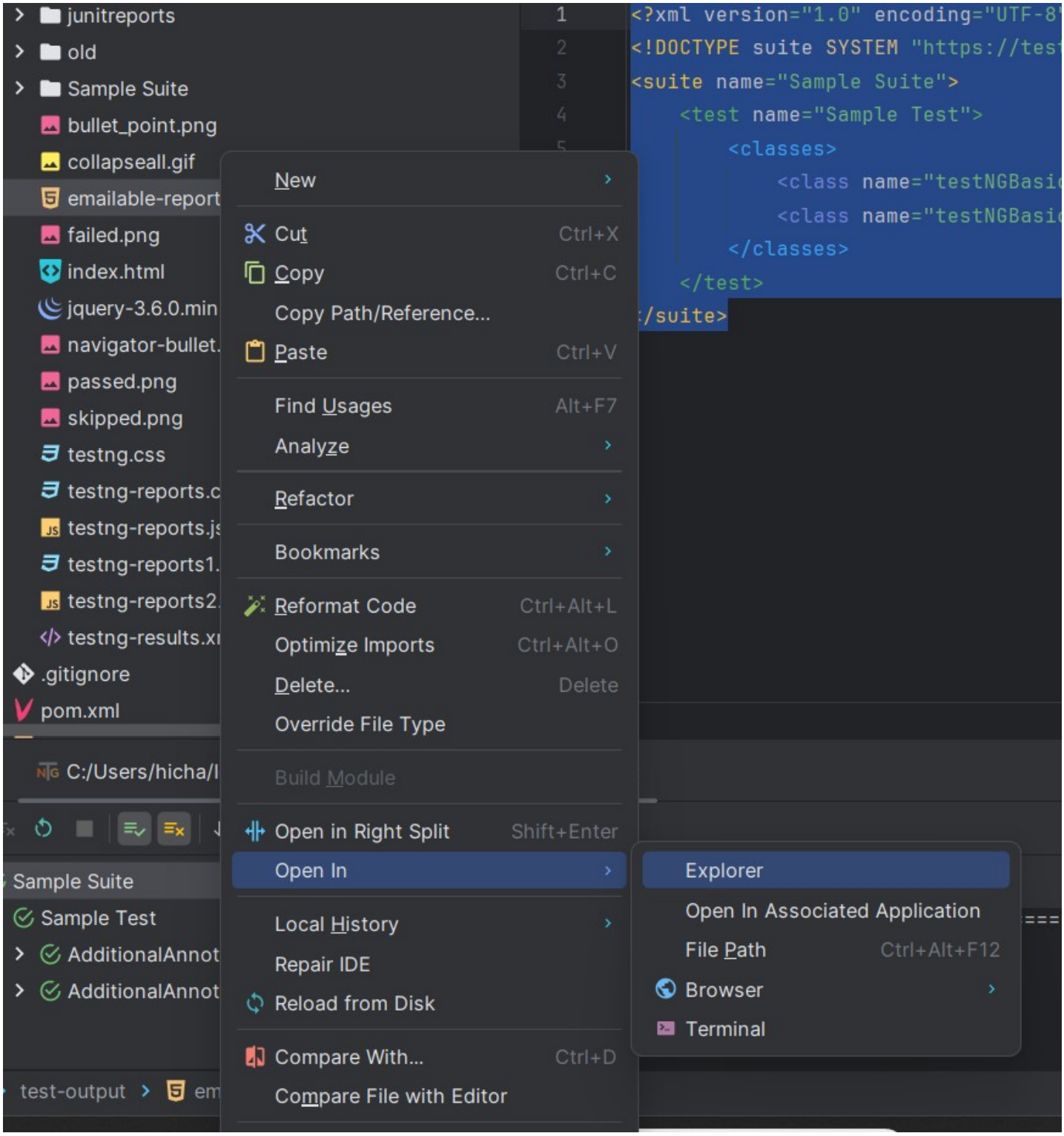
```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Sample Suite">
    <test name="Sample Test">
        <classes>
            <class name="testNGBasics.AdditionalAnnotationsOneTest"/>
            <class name="testNGBasics.AdditionalAnnotationsTwoTest"/>
        </classes>
    </test>
</suite>
```

How to open the emailable-report.html and index.html ?

Right Click on them, then open them by using:
Explorer.
Browser.

How share the files ?

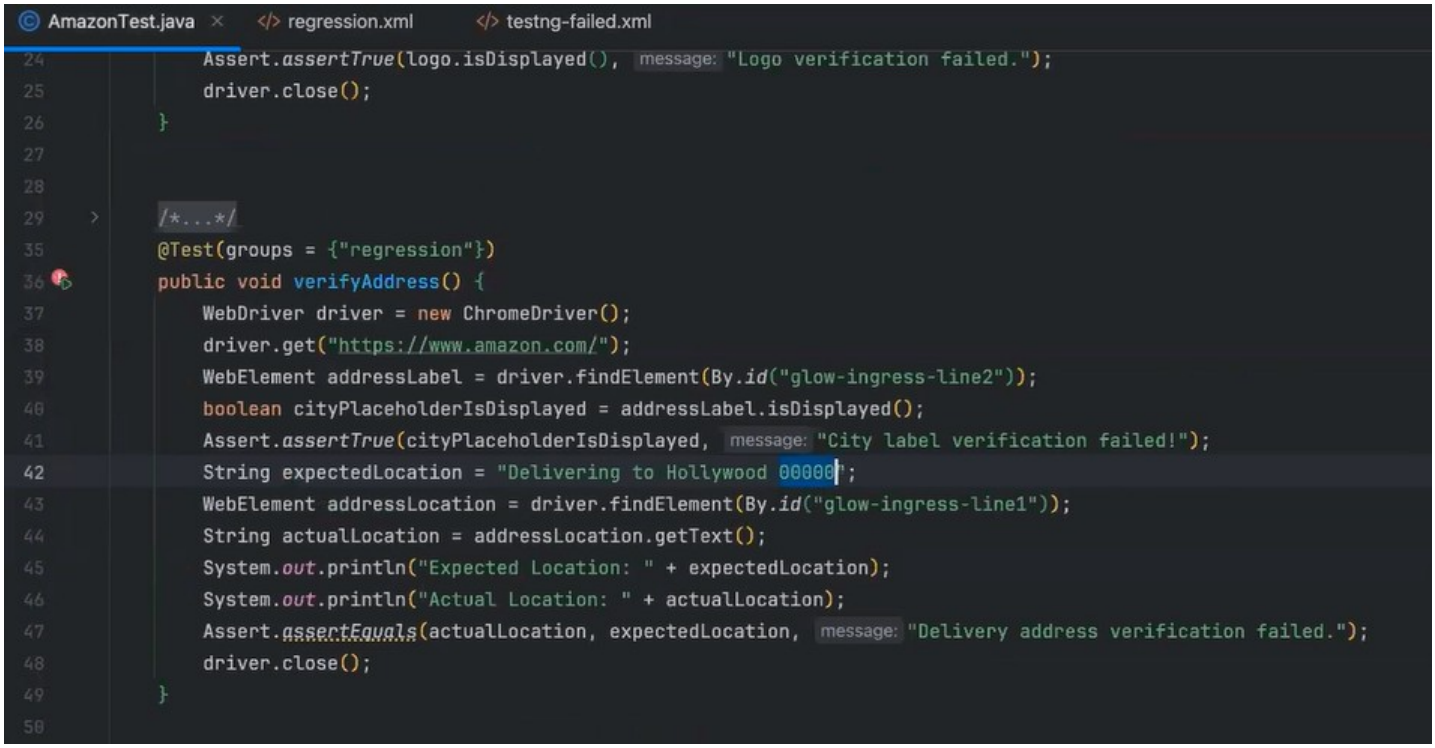
By using Jenkins.

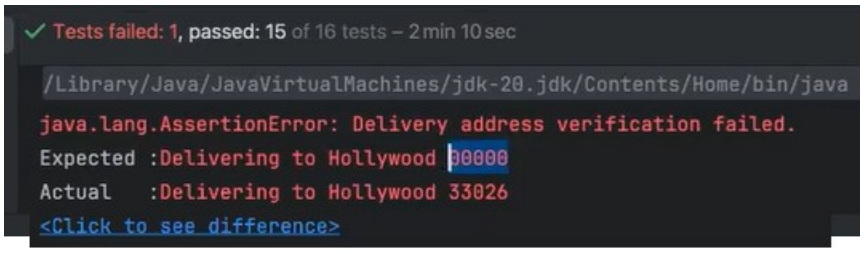

junitreports
old
Sample Suite
bullet_point.png
collapseall.gif
emailable-report
failed.png
index.html
jquery-3.6.0.min
navigator-bullet.
passed.png
skipped.png
testng.css
testng-reports.c
testng-reports.js
testng-reports1.
testng-reports2.
testng-results.xr
.gitignore
pom.xml
C:/Users/hicha/I
Sample Suite
Sample Test
AdditionalAnnot
AdditionalAnnot
test-output
em
1
2
3
4
<?xml version="1.0" encoding="UTF-8"
<!DOCTYPE suite SYSTEM "https://test
<suite name="Sample Suite">
<test name="Sample Test">
<classes>
<class name="testNGBasic
<class name="testNGBasic
</classes>
</test>
</suite>
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Override File Type
Build Module
Open in Right Split Shift+Enter
Open In
Local History
Repair IDE
Reload from Disk
Compare With... Ctrl+D
Compare File with Editor
Explorer
Open In Associated Application
File Path Ctrl+Alt+F12
Browser
Terminal

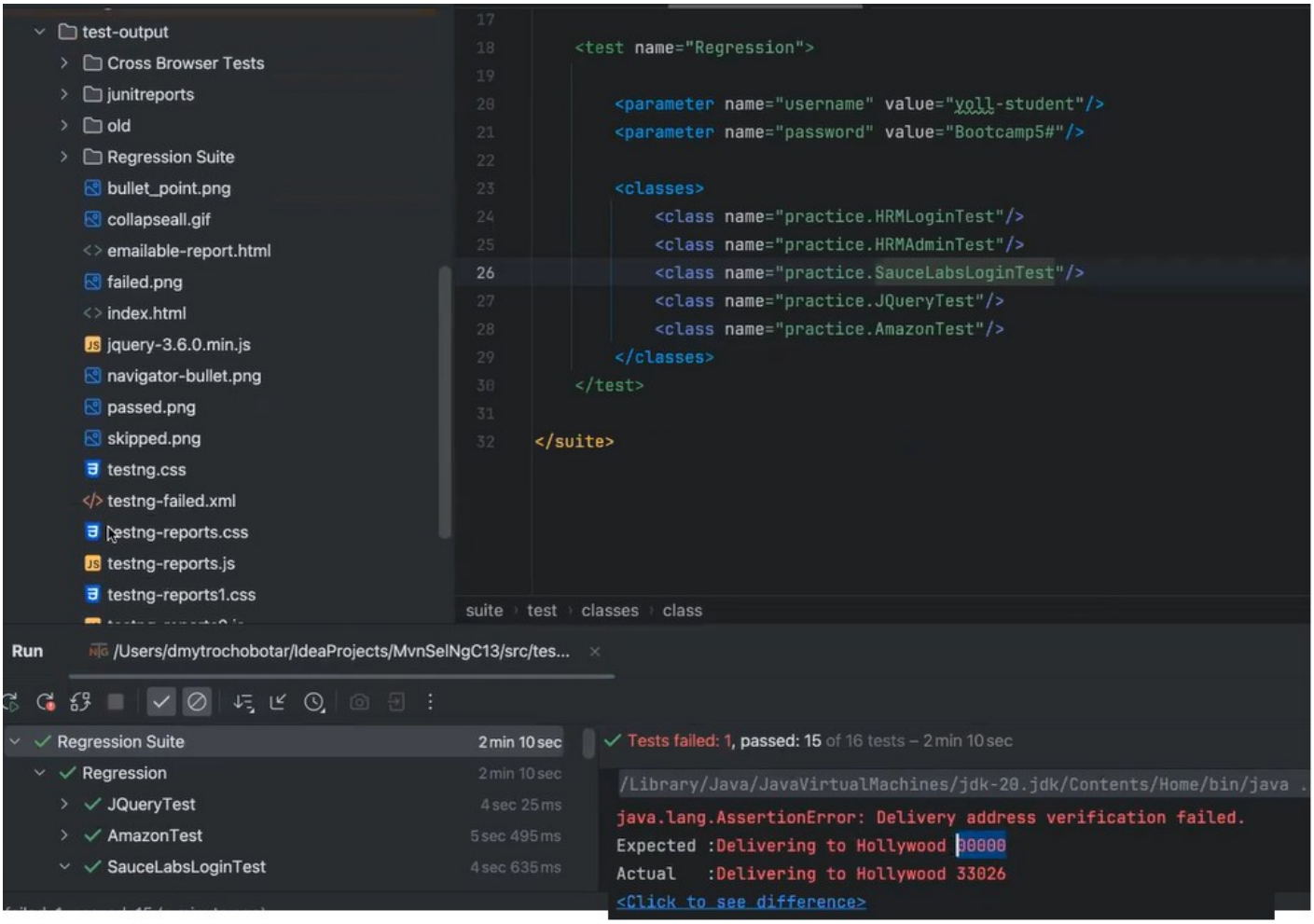
Testng-failed.xml

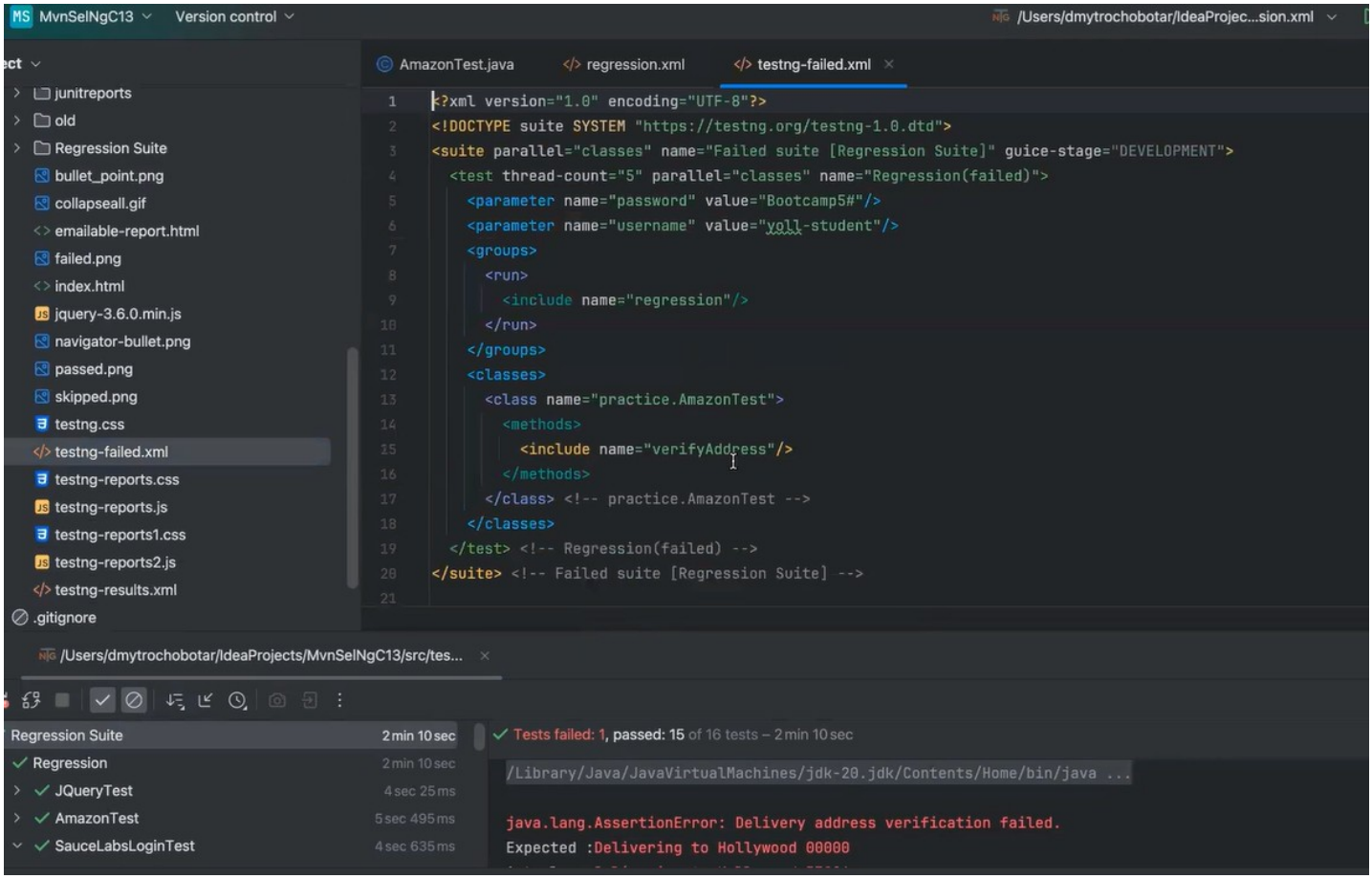
When there is a failed test, we can see it from this xml file: Testng-failed.xml

In our case, we run regression.xml file that has a failed test. (Testng-failed.xml) helps to fix it easily.




Tests failed: 1, passed: 15 of 16 tests – 2 min 10 sec
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java
java.lang.AssertionError: Delivery address verification failed.
Expected :Delivering to Hollywood 00000
Actual :Delivering to Hollywood 33026
<Click to see difference>


test-output
Cross Browser Tests
junitreports
old
Regression Suite
bullet_point.png
collapseall.gif
emailable-report.html
failed.png
index.html
jquery-3.6.0.min.js
navigator-bullet.png
passed.png
skipped.png
testng.css
testng-failed.xml
testng-reports.css
testng-reports.js
testng-reports1.css
<test name="Regression">
<parameter name="username" value="yoll-student"/>
<parameter name="password" value="Bootcamp5#"/>
<classes>
<class name="practice.HRMLoginTest"/>
<class name="practice.HRMAdminTest"/>
<class name="practice.SauceLabsLoginTest"/>
<class name="practice.JQueryTest"/>
<class name="practice.AmazonTest"/>
</classes>
</test>
</suite>
suite › test › classes › class
Run
/Users/dmytrochobotar/IdeaProjects/MvnSelNgC13/src/tes...
Regression Suite 2 min 10 sec
Regression 2 min 10 sec
JQueryTest 4 sec 25 ms
AmazonTest 5 sec 495 ms
SauceLabsLoginTest 4 sec 635 ms
Tests failed: 1, passed: 15 of 16 tests – 2 min 10 sec
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java .
java.lang.AssertionError: Delivery address verification failed.
Expected :Delivering to Hollywood 00000
Actual :Delivering to Hollywood 33026
<Click to see difference>



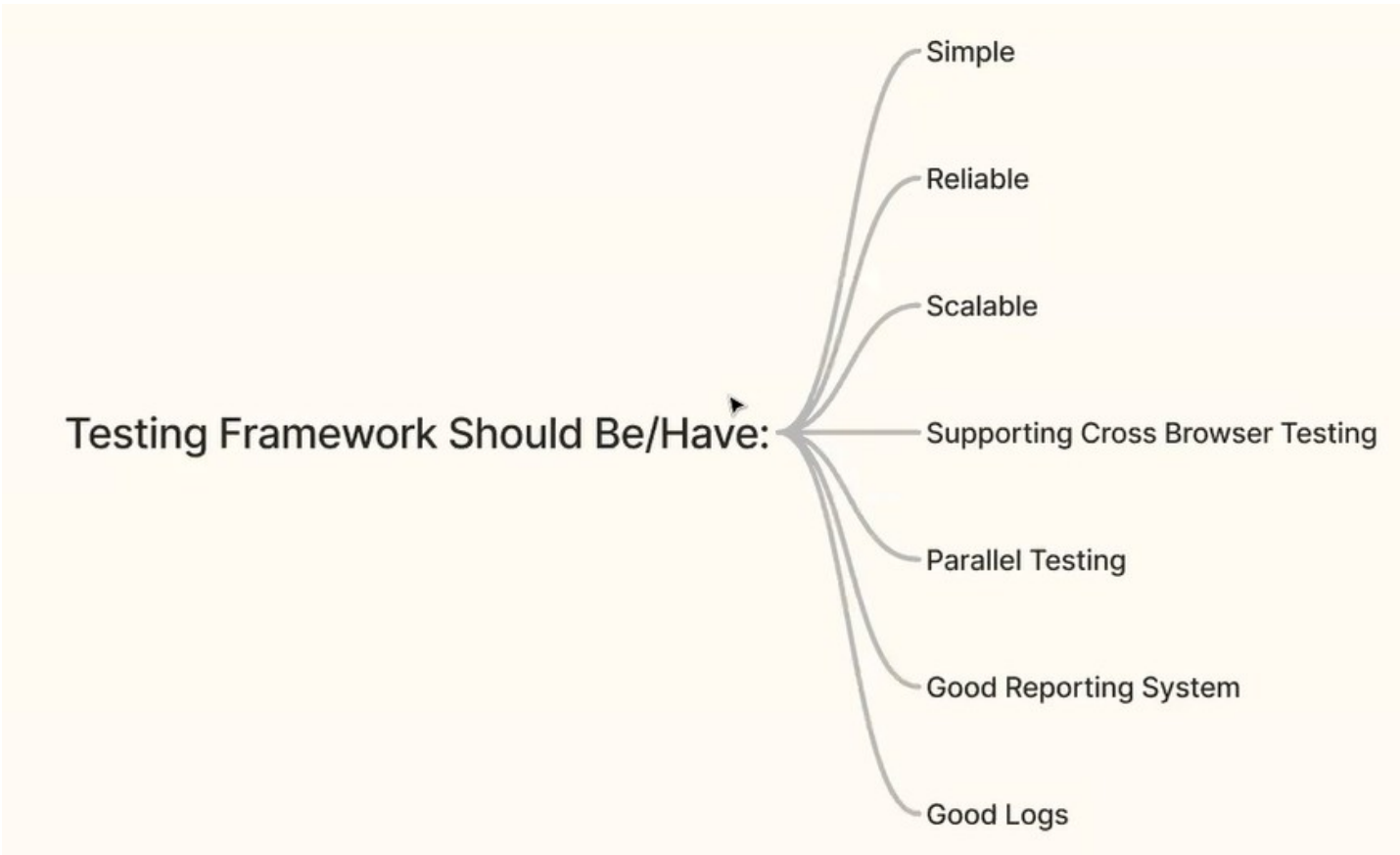
FUNCTIONALITY: TESTNG VS JUNIT

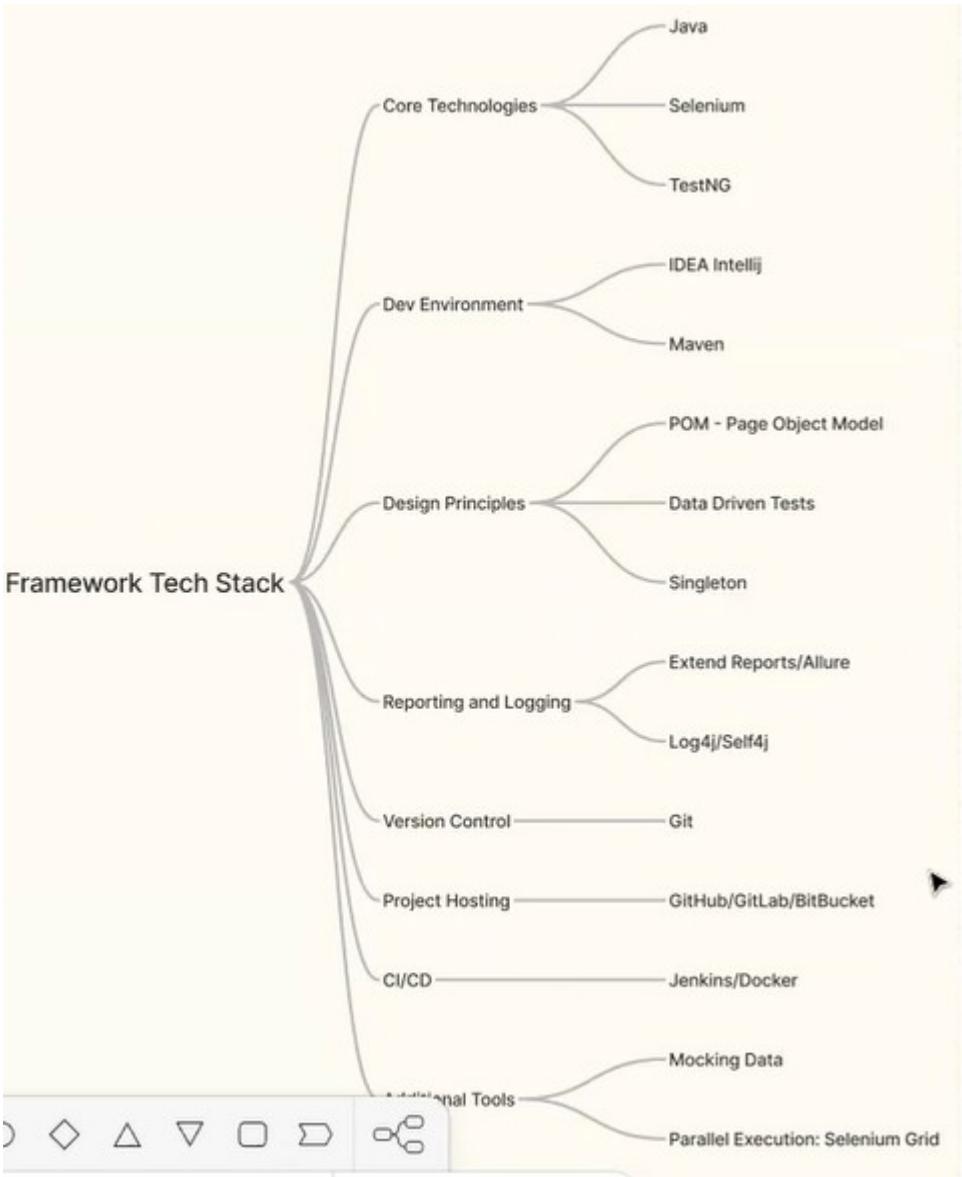
Functionality - JUnit 4 vs TestNG

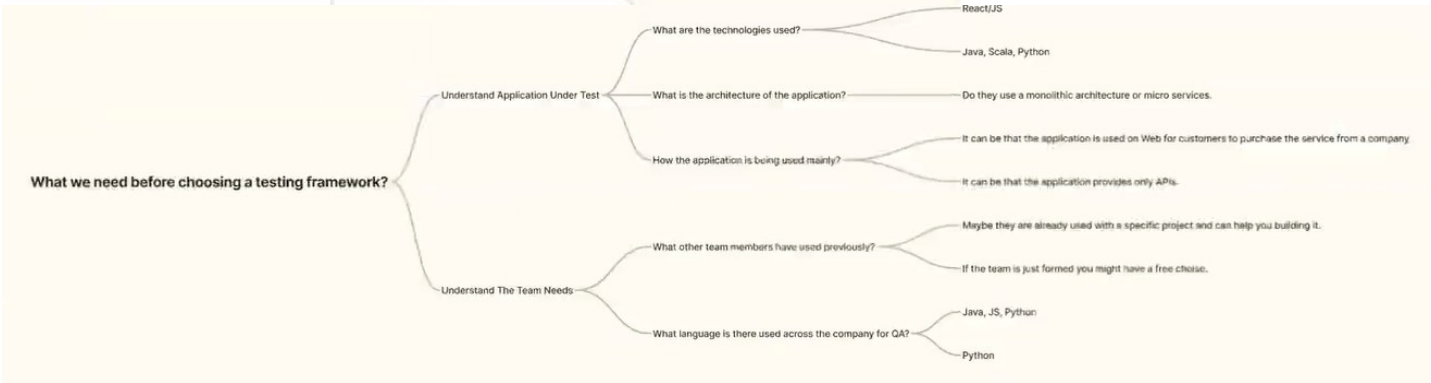
	Annotation Support	Exception Test	Ignore Test	Timeout Test	Suite Test	Group Test	Parameterized (primitive value)	Parameterized (object)	Dependency Test
TestNG	✅	✅	✅	✅	✅	✅	✅	✅	✅
JUnit 4	✅	✅	✅	✅	✅	❌	✅	❌	❌

- TestNG is more advanced in **parameterized testing, dependency testing**.
- TestNG has its own html reporter, also creates separate report for failed test cases.
- TestNg provides testng xml files to execute different level of tests.(suite level, class level)
- Integrating multiple tests together and then **running them in parallel** as a test suite used to be a feature available for only TestNG.
    - Junit 4 and Junit 5 requires maven surefire plugins and extra tools like cucumber to create html report.

- Some of the feature TestNG has, Junit5 is catching up. But still, those tasks are simpler to do in TestNG.


Testing Framework Should Be/Have:
Simple
Reliable
Scalable
Supporting Cross Browser Testing
Parallel Testing
Good Reporting System
Good Logs

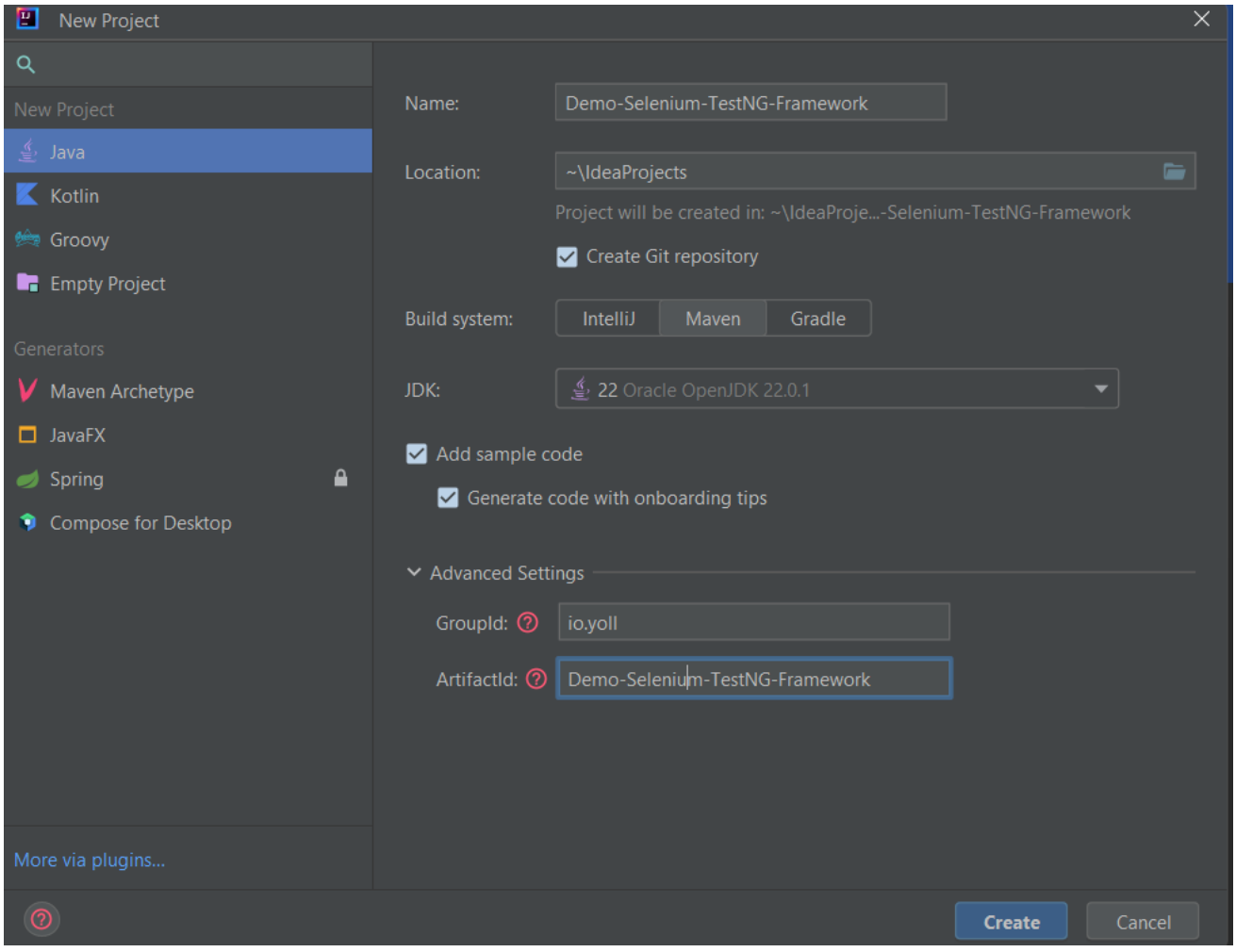

Framework Tech Stack
Core Technologies
Java
Selenium
TestNG
Dev Environment
IDEA Intellij
Maven
Design Principles
POM - Page Object Model
Data Driven Tests
Singleton
Reporting and Logging
Extend Reports/Allure
Log4j/Self4j
Version Control
Git
Project Hosting
GitHub/GitLab/BitBucket
CI/CD
Jenkins/Docker
Mocking Data
Parallel Execution: Selenium Grid


What we need before choosing a testing framework?
Understand Application Under Test
What are the technologies used?
React/JS
Java, Scala, Python
What is the architecture of the application?
Do they use a monolithic architecture or micro services.
How the application is being used mainly?
It can be that the application is used on Web for customers to purchase the service from a company.
It can be that the application provides only APIs.
Understand The Team Needs
What other team members have used previously?
Maybe they are already used with a specific project and can help you building it.
If the team is just formed you might have a free choise.
What language is there used across the company for QA?
Java, JS, Python
Python

Creating Testing Framework:

New Project / With Create Git Repository / Maven
Group id : starts with domain then
com.orangeHrm

Yoll.io >> io.yoll
Amazon.com>>com.amazon
Artifactid : is the same name as the project name

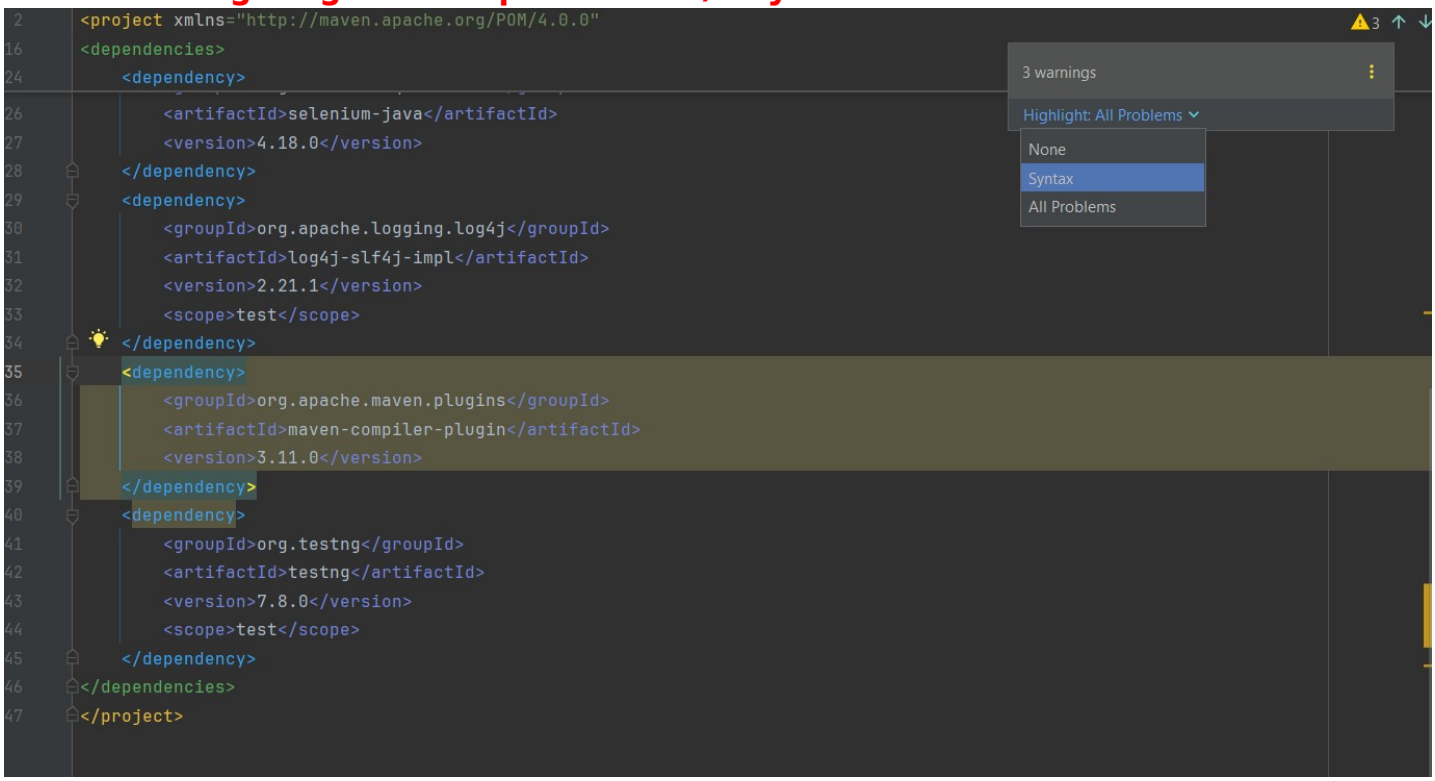


Creating README.md

New File / write : README.md / Add

Create the description from chatgpt
Create resources folder under test folder:
new/Directory
Add dependencies on pom.xml file

If we have warnings, click on the yellow sign, and choose Highlight All problems / Syntax

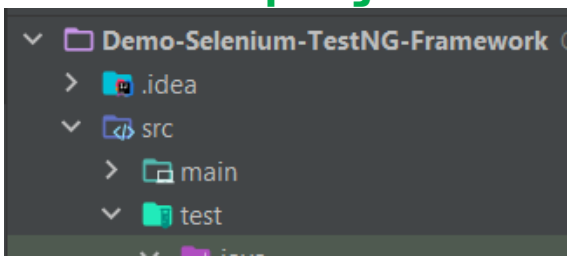



```
<project xmlns="http://maven.apache.org/POM/4.0.0"
<dependencies>
    <dependency>
        <artifactId>selenium-java</artifactId>
        <version>4.18.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>2.21.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11.0</version>
    </dependency>
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.8.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
</project>
```

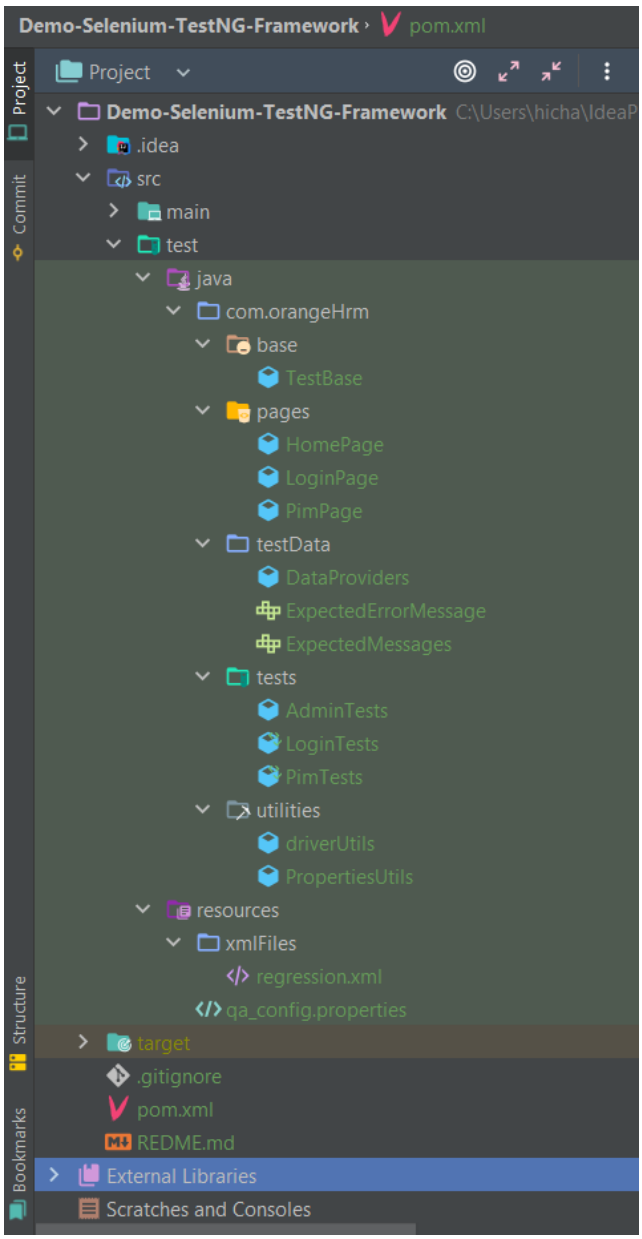
Then we create 3 packages:
First level under java folder: as an example
com.orangeHrm
Io.yoll
Second level under java folder:

for Developemnt Environement:
dev-hrm.yoll.io
for Production Environement:
orangehrm.com
Io.yoll

the Java project name must have the company name,



but under java folder, we use the project in the reversed way (com.orangeHrm) (Io.yoll).



And under com.orangeHrm, create these packages:

- **base, By entering this : com.orangeHrm.base**
  Under base, create TestBase class.
- **pages, By entering this : com.orangeHrm.pages**
  Under pages, create HomePage, LoginPage, PimPage classes.
- **testData, By entering this : com.orangeHrm.testData**

Under testData, create DataProviders class,ExpectedErrorMessage and ExpectedErrorMessage interfaces.

- tests, By entering this : com.orangeHrm.tests
  Under pages, create AdminTests, LoginTests, PimTest classes.

- utilities, By entering this : com.orangeHrm.utilities
  Under pages, create driverUtiles and PropertiesUtils classes.

Choose YES to Add the classes to Git.

Under resources Folder, create
- xmlFiles Folder that contains regression.xml file.
- qa_config.properties. It is a file with .properties extension

```
package com.orangeHrm.pages;
If we have id on the html, we can just use id for each element !
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage {
    public LoginPage(WebDriver driver){
        PageFactory.initElements(driver, this);
    }
    @FindBy(id = "txtUsername")
    private WebElement userNameInputBox;

      @FindBy(id = "txtPassword")
    //@FindBy(xpath = "//*[@id='txtPassword']")
    private WebElement passWordInputBox;

    //@FindBy(css="input#btnLogin")
```

```
    @FindBy(id="input#btnLogin")
    private WebElement loginButton;


    public void login(String userName, String passWord) {
        userNameInputBox.sendKeys(userName);
        passWordInputBox.sendKeys(passWord);
        loginButton.click();

    }
}
```

This code is a typical implementation of the **Page Object Model (POM)** design pattern in Selenium WebDriver. It encapsulates the elements and behaviors of a login page into a class, making the test scripts more organized, maintainable, and reusable.

Explanation of Each Part:

1. Package Declaration:

```
package com.orangeHrm.pages;
```

- This line declares that the class is part of the `com.orangeHrm.pages` package. This helps in organizing the classes logically, such as putting all page-related classes under a `pages` package.

2. Import Statements:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
```

- `WebDriver`: This is the main interface for interacting with the web browser in Selenium.
- `WebElement`: Represents an element on a webpage. All the actions, such as clicking or sending keys, are performed on `WebElement` objects.
- `FindBy`: An annotation used to locate elements on the webpage. It supports various locators like `id`, `xpath`, and `css`.
- `PageFactory`: A Selenium class used to initialize the WebElements within a page object.

3. Class Declaration:

```
public class LoginPage {
```

- This class, `LoginPage`, represents the login page of the OrangeHRM application. The class contains methods and elements specific to the login functionality of the page.

4. Constructor:

```
public LoginPage(WebDriver driver){
    PageFactory.initElements(driver, this);
}
```

- **Purpose**: This constructor initializes the web elements defined within the `LoginPage` class using the `PageFactory.initElements()` method.
  - `driver`: The `WebDriver` instance that interacts with the browser.
  - `PageFactory.initElements(driver, this)` binds the driver to the web elements defined in the class using `@FindBy` annotations. This allows you to interact with elements without explicitly calling `driver.findElement()` for each one.

5. WebElement Definitions:

```
@FindBy(id = "txtUsername")
private WebElement userNameInputBox;

@FindBy(xpath = "//*[@id='txtPassword']")
private WebElement passWordInputBox;

@FindBy(css="input#btnLogin")
private WebElement loginButton;
```

- These lines define the elements on the login page that the test will interact with, using the `@FindBy` annotation to locate the elements on the webpage:
  - **`userNameInputBox`**: Represents the username input field, identified by the `id="txtUsername"`.
  - **`passWordInputBox`**: Represents the password input field, located using an XPath expression.
  - **`loginButton`**: Represents the login button, identified using a CSS selector (`input#btnLogin`).

These fields are marked `private` to encapsulate them within the class, providing controlled access through methods.

6. `login` Method:

```
    public void login(String userName, String passWord) {
    userNameInputBox.sendKeys(userName);
    passWordInputBox.sendKeys(passWord);
    loginButton.click();
}
```

- **Purpose**: This method encapsulates the behavior of logging in by performing three actions:
    1. **Enter Username**: `userNameInputBox.sendKeys(userName)` sends the provided `userName` to the username input field.
    2. **Enter Password**: `passWordInputBox.sendKeys(passWord)` sends the provided `passWord` to the password input field.
    3. **Click Login Button**: `loginButton.click()` clicks the login button to submit the login form.
- **Parameters**:

    1. `userName`: The username to be entered in the username field.
    2. `passWord`: The password to be entered in the password field.

Benefits of This Design:

1. **Reusability**: The `LoginPage` class can be reused across multiple test cases that involve logging in, without repeating code.
2. **Maintainability**: If there is a change in the login page (e.g., element locators change), the test logic won't need to be modified. Only the locators in the `LoginPage` class would need to be updated.
3. **Readability**: The test scripts using this class will be easier to read since the details of interacting with web elements are abstracted into the `LoginPage` class.
4. **Encapsulation**: The `LoginPage` class encapsulates both the elements and actions related to the login functionality, making the code more modular.

TestNG Framework (Final Version)

TestBase class:

```
package com.orangeHrm.base;

import com.orangeHrm.utilities.PropertiesUtils;
import com.orangeHrm.utilities.driverUtils;
import org.openqa.selenium.WebDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Parameters;

import java.time.Duration;

public class TestBase {
    protected WebDriver driver;
    @Parameters("browserType")
    @BeforeMethod
    public void setUp(String browserType){
        driver = driverUtils.getDriver(browserType);
        driver.manage().window().maximize();
```

```
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
        driver.get(PropertiesUtils.getUrl());
    }
    @AfterMethod
    public void tearDown(){
        driver.quit();
    }
}   package com.orangeHrm.pages;
```

HomePage Class:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class HomePage {
    public HomePage(WebDriver driver){
        PageFactory.initElements(driver, this);
    }
    @FindBy(id="welcome")
    private WebElement welcomePage;
    @FindBy(id = "menu_pim_viewPimModule")
    private WebElement PimTab;
    public String getWelcomeMessageText(){
        return welcomePage.getText();
    }
    public void PimTabVerification(){
        PimTab.click();
    }
}
```

LoginPage Class

```
package com.orangeHrm.pages;


import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage {
    public LoginPage(WebDriver driver){
        PageFactory.initElements(driver, this);
    }
    @FindBy(id = "txtUsername")
    private WebElement userNameInputBox;
    @FindBy(id = "txtPassword")
    private WebElement passWordInputBox;
    @FindBy(id = "btnLogin")
    private WebElement loginButton;
    @FindBy(id = "spanMessage")
    private WebElement invalidCredentialsErrorMessage;
    public void login(String username, String password){
```

```
        userNameInputBox.sendKeys(username);
        passWordInputBox.sendKeys(password);
        loginButton.click();

    }
    public void loginWithInvalidCredentials(String username, String password){
        userNameInputBox.sendKeys(username);
        passWordInputBox.sendKeys(password);
        loginButton.click();
    }
    public String InvalidCredentialsErrorMessage(){
        return invalidCredentialsErrorMessage.getText();
    }
}
```

PimPage Class:

```
 package com.orangeHrm.pages;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class PimPage {
    public PimPage(WebDriver driver){
        PageFactory.initElements(driver,this);
    }

    @FindBy(className= "head")
    private WebElement PimPageHeader;
    @FindBy(id="menu_pim_addEmployee")
    private WebElement addEmployeeTab;
    @FindBy(id="firstName")
    private WebElement firstNameText;
    @FindBy(id="lastName")
    private WebElement lastNameText;
    @FindBy(id="chkLogin")
    private WebElement createLoginDetailsButton;
    @FindBy(id="user_name")
    public WebElement userName;
    @FindBy(xpath="//*[@name='user_password']")
    public WebElement passWord;



    public String PimPageHeaderVerification(){
        return PimPageHeader.getText();
    }
    public void AddEmployeeFields() throws InterruptedException {
        addEmployeeTab.click();
        Thread.sleep(1500);
        firstNameText.sendKeys("Meryem");
        Thread.sleep(1500);
        lastNameText.sendKeys("Chehdaoui");
        Thread.sleep(1500);
```

```
        createLoginDetailsButton.click();

    }


}
```

DataProviders Class:

```
package com.orangeHrm.testData;



import org.testng.annotations.DataProvider;

public class DataProviders {
    @DataProvider(name = "Invalid Login Credentials")
    public Object[][] getInvalidCredentials(){
        Object[][] obj = new Object[][]{
                {"invalid","invalid",ExpectedErrorMessage.INVALID_CREDENTIAL},
                {"","invalid",ExpectedErrorMessage.EMPTY_USERNAME},
                {"invalid","",ExpectedErrorMessage.EMPTY_PASSWORD}
        };
        return obj;
    }


}
```

ExpectedErrorMessage interface:

```
package com.orangeHrm.testData;

public interface ExpectedErrorMessage {
    String INVALID_CREDENTIAL = "Invalid credentials";
    String EMPTY_USERNAME = "Username cannot be empty";
    String EMPTY_PASSWORD = "Password cannot be empty";
}
```

ExpectedMessages interface:

```
package com.orangeHrm.testData;

public interface ExpectedMessages {
    String WELCOME_MESSAGE = "Welcome Yoll";
    String PIM_TAB_HEADER = "Employee Information";
}
```

AdminTests extends TestBase class:

```
package com.orangeHrm.tests;
import com.orangeHrm.base.TestBase;

public class AdminTests extends TestBase{
}
```

LoginTests extends TestBase class:

```
package com.orangeHrm.tests;
import com.orangeHrm.testData.DataProviders;
import com.orangeHrm.testData.ExpectedMessages;
import com.orangeHrm.base.TestBase;
import com.orangeHrm.pages.HomePage;
import com.orangeHrm.pages.LoginPage;
import com.orangeHrm.utilities.PropertiesUtils;
import org.testng.annotations.*;
import org.testng.Assert;

public class LoginTests extends TestBase{

    @Test
    public void testLoginWithValidCredentials(){
        LoginPage loginPage = new LoginPage(driver);
        loginPage.login(PropertiesUtils.getUsername(),PropertiesUtils.getPassword());
        HomePage homepage = new HomePage(driver);
        String actualMessage = homepage.getWelcomeMessageText();
        Assert.assertEquals(actualMessage, ExpectedMessages.WELCOME_MESSAGE,"welcome
message verification failed.");
    }
    @Test(dataProviderClass = DataProviders.class, dataProvider = "Invalid Login
Credentials")
    public void testLoginWithInvalidCredentials(String username, String password, String
errorMessage){
        LoginPage loginPage = new LoginPage(driver);
        loginPage.login(username, password);
        Assert.assertEquals(loginPage.InvalidCredentialsErrorMessage(), errorMessage,"the
error message failed.");
    }
}
```

PimTests extends TestBase class:

```
package com.orangeHrm.tests;

import com.orangeHrm.testData.ExpectedMessages;
import com.orangeHrm.base.TestBase;
import com.orangeHrm.pages.HomePage;
import com.orangeHrm.pages.LoginPage;
import com.orangeHrm.pages.PimPage;
import com.orangeHrm.utilities.PropertiesUtils;
import org.testng.Assert;
import org.testng.annotations.Test;

public class PimTests extends TestBase {

    @Test
    public void PimTabAccess(){
        LoginPage loginPage = new LoginPage(driver);
        HomePage homePage = new HomePage(driver);
        PimPage pimpage = new PimPage(driver);
```

```
        loginPage.login(PropertiesUtils.getUsername(),PropertiesUtils.getPassword());
        homePage.PimTabVerification();
        String actualPimHeader = pimpage.PimPageHeaderVerification();
        Assert.assertEquals(actualPimHeader, ExpectedMessages.PIM_TAB_HEADER,"Pim Header
is failed");
    }
}
```

driverUtils Class

```
package com.orangeHrm.utilities;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.safari.SafariDriver;
public class driverUtils {
    public static WebDriver getDriver(String browserType){
        return switch(browserType){
            case"fireFox" -> new FirefoxDriver();
            case"safari" ->  new SafariDriver();
            default -> new ChromeDriver();
        };
    }
}
```

PropertiesUtils class

```
package com.orangeHrm.utilities;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
public class PropertiesUtils {
    private static Properties properties;
    public static void loadPropetyFile() throws IOException {
        properties = new Properties();
        FileInputStream stream = new
FileInputStream("src/test/resources/qa_config.properties");
        properties.load(stream);
    }
    public static String getUsername(){
        try {
            loadPropetyFile();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return properties.getProperty("username");
    }
    public static String getPassword(){
        try {
            loadPropetyFile();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return properties.getProperty("password");
    }
    public static String getUrl(){
        try {
            loadPropetyFile();
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return properties.getProperty("baseUrl");
    }
}
```

Regression.xml file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="regression suite">
    <test name="regression test">
        <parameter name="browserType" value="chrome"/>
        <packages>
            <package name="com.orangeHrm.tests"/>
        </packages>
    </test>
</suite>   in qa_config.properties file :
```

qa_config.properties file :

```
username =yoll-student
password = Bootcamp5#
baseUrl = http://dev-hrm.yoll.io/
```

Explaination:

This set of code is a complete framework using the **Page Object Model (POM)** pattern along with **TestNG** as a test runner for testing an Orange HRM (Human Resource Management) web application. Below, I'll provide a detailed explanation of each part.

1. `TestBase` Class (`com.orangeHrm.base.TestBase`)

- The `TestBase` class serves as a base setup and teardown configuration for all tests.
- It has a protected `WebDriver driver` variable to be used in child classes.
- The `@BeforeMethod` and `@AfterMethod` annotations are used to set up and tear down the browser before and after each test.
- `setUp(String browserType)`: Initializes the WebDriver, maximizes the browser window, sets an implicit wait, and navigates to the URL retrieved from the `PropertiesUtils` class.
- `tearDown()`: Closes the browser after each test.
- `@Parameters("browserType")`: Allows passing the `browserType` parameter through the XML file for cross-browser testing.

2. `LoginPage` Class (`com.orangeHrm.pages.LoginPage`)

- Represents the login page in the Orange HRM application.
- Uses `@FindBy` annotations to locate UI elements (`userNameInputBox`, `passWordInputBox`, `loginButton`, and `invalidCredentialsErrorMessage`).
- Provides reusable methods for interacting with these elements (`login()` and `loginWithInvalidCredentials()`).
  - `login(String username, String password)`: Takes the username and password as arguments and logs in.
  - `InvalidCredentialsErrorMessage()`: Returns the error message text displayed when credentials are invalid.

3. `HomePage` Class (`com.orangeHrm.pages.HomePage`)

- Represents the home page after a successful login.
- Locates elements (`welcomePage` and `PimTab`) using `@FindBy`.
- Provides methods to interact with the elements:
  - `getWelcomeMessageText()`: Returns the welcome message text.
  - `PimTabVerification()`: Clicks on the PIM (Personal Information Management) tab to navigate to the PIM section.

4. `PimPage` Class (`com.orangeHrm.pages.PimPage`)

- Represents the PIM page for adding or managing employee information.
- Locates elements (`PimPageHeader`, `addEmployeeTab`, `firstNameText`, `lastNameText`, `createLoginDetailsButton`, etc.) using `@FindBy`.
- Provides methods for interacting with the PIM page:
  - `PimPageHeaderVerification()`: Returns the header text of the PIM page.
  - `AddEmployeeFields()`: Adds an employee by filling out `firstName`, `lastName`, and clicking `createLoginDetailsButton`.

5. `DataProviders` Class (`com.orangeHrm.testData.DataProviders`)

- This class provides a data set to be used in data-driven tests with the `@DataProvider` annotation.
- `getInvalidCredentials()`: Provides a set of invalid login credentials along with expected error messages (`INVALID_CREDENTIAL`, `EMPTY_USERNAME`, `EMPTY_PASSWORD`).

- `@DataProvider(name = "Invalid Login Credentials")`: This is used to supply multiple sets of data for the login test cases.

6. `ExpectedErrorMessage` and `ExpectedMessages` Interfaces (`com.orangeHrm.testData`)

- These interfaces hold constant values for error messages and expected messages used in assertions.
  - `INVALID_CREDENTIAL`, `EMPTY_USERNAME`, `EMPTY_PASSWORD`: Possible error messages when login credentials are invalid.
  - `WELCOME_MESSAGE`, `PIM_TAB_HEADER`: Expected messages for successful navigation.

7. `driverUtils` Class (`com.orangeHrm.utilities.driverUtils`)

- The utility class for handling WebDriver instantiation.
- The `getDriver(String browserType)` method uses a switch expression to return different `WebDriver` instances (Chrome, Firefox, Safari) based on the browser type specified.
  - This enables cross-browser testing.

8. `PropertiesUtils` Class (`com.orangeHrm.utilities.PropertiesUtils`)

- A utility class for reading properties from the configuration file (`qa_config.properties`).
- Methods to load properties and return values (`getUsername()`, `getPassword()`, `getUrl()`).
- Loads the property file (`qa_config.properties`) containing details like `username`, `password`, and `baseUrl`.

9. `LoginTests` Class (`com.orangeHrm.tests.LoginTests`)

- Extends `TestBase` to inherit the setup and teardown.
- Contains test methods:
  - `testLoginWithValidCredentials()`: Tests login functionality using valid credentials. Asserts the welcome message on the home page.
  - `testLoginWithInvalidCredentials()`: Tests login using invalid credentials, using a data provider. Asserts the error messages.

10. `PimTests` Class (`com.orangeHrm.tests.PimTests`)

- Also extends `TestBase` and contains tests related to the PIM section.
- `PimTabAccess()`: Tests navigation to the PIM page after login. Asserts that the PIM page header matches the expected header.

11. TestNG Suite (`.xml` file)

- This XML file (`suite`) configures the entire test suite for TestNG.
- Defines a suite called `"regression suite"` and a test called `"regression test"`.
- Parameter `"browserType"` is specified to use Chrome by default.
- Uses the `com.orangeHrm.tests` package to run all the test classes within.

12. `qa_config.properties` file

- This properties file stores values required for the tests, such as `username`, `password`, and `baseUrl`.
- These values are read by the `PropertiesUtils` utility to provide configuration details for the tests.

Summary:

- This code demonstrates a typical **Page Object Model (POM)** pattern for an automated test framework using Selenium WebDriver and TestNG.
- **`TestBase` Class** provides common setup and teardown functionality.
- **Page Classes (`LoginPage`, `HomePage`, `PimPage`)** contain web element locators and reusable methods for interacting with pages, promoting **modularity** and **maintainability**.
- **Test Classes (`LoginTests`, `PimTests`)** focus on actual test scenarios, making them more **readable**.
- **Utility Classes (`driverUtils`, `PropertiesUtils`)** abstract repetitive tasks like driver management and configuration, making the test code **cleaner**.
- **Data Providers** are used to provide multiple sets of input data for data-driven testing.

This modular and well-structured approach makes it easier to maintain and extend the testing framework.

Waiting
First way : using Thread.*sleep*();

```
/*
In reality the code is much faster then web application rendering/loading
We need to take care in our code about synchronizing it
* The approach we have used is Thread.sleep()
*This means we visually would identify where we need to stop execution of our
code
And apply the wait there
* However, there is no guarantee it will work all the time
*Also, if the element can be located faster, code will still be paused for that
amount of time *This leads to an increased time of execution of the scripts
**/
```

Open https://www.saucedemo.com/ using username, password, and click on login button
Go to : backpackTitle, and click on the button
Go Back to the main page.
Go to : bikeLightTitle, and click on the button
Go Back to the main page.

This code has duplicate actions, and unnecessary Thread.sleep(). Instead of using Thread.sleep(), it's better to use Selenium waits for more reliability.

```
package waitPractice;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.Test;

public class PracticeWaits {
    @Test
    public void testCase() throws InterruptedException {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.saucedemo.com/");
        WebElement usernameInputBox = driver.findElement(By.id("user-name"));
        WebElement passwordInputBox = driver.findElement(By.id("password"));
        WebElement loginButton = driver.findElement(By.id("login-button"));
        usernameInputBox.sendKeys("performance_glitch_user");
        passwordInputBox.sendKeys("secret_sauce");
        loginButton.click();
        Thread.sleep(5000);
        WebElement backpackTitle = driver.findElement(By.xpath("//*[text() = 'Sauce Labs
Backpack']"));
        backpackTitle.click();
        Thread.sleep(5000);
        WebElement backToHomePage = driver.findElement(By.id("back-to-products"));
```

```
        backToHomePage.click();
        Thread.sleep(5000);

        WebElement bikeLightTitle = driver.findElement(By.xpath("//*[text() = 'Sauce Labs
Bike Light']"));
        bikeLightTitle.click();
        Thread.sleep(5000);

        backToHomePage = driver.findElement(By.id("back-to-products"));
        backToHomePage.click();
      // Close the driver
      driver.quit();

    }}
```

Second way : using Implicit Wait

- Implicit wait is a time limit we provide for Selenium to wait, before throwing elementNotFound exception.
- Implicit wait is dynamic
- That means, if we setup our implicit wait to 8 seconds, and element was found after 2 seconds, Implicit Wait will move on to next step, and will not wait whole 8 seconds.
  - `driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(8));`
- This is a huge improvement compared to sleep.
- However implicit wait has limitations
  - The implicit wait is set only once and lasts for the **entire duration of your webdriver**.(= scope entire tests)
  - That means we just declared for **all elements** to wait up to 8 seconds.
  - What if some elements take longer to load?
  - What if it is acceptable for some element to load in 5 seconds, while it is not ok acceptable for others?

Implicit waits tell Selenium how long to wait for each element to become present in the DOM (Document Object Model) when trying to find it before throwing a `NoSuchElementException`.
If the element is found before the 10-second period ends, WebDriver will proceed immediately without waiting for the full duration.

How Implicit Wait Works

- Suppose you set an implicit wait of 10 seconds: driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));.

- When you try to find an element with driver.findElement(), WebDriver will wait up to 10 seconds for the element to appear in the DOM.
- If the element is found within, say, 3 seconds, WebDriver immediately proceeds without waiting further.
- If the element is not found within the 10 seconds, Selenium will throw a NoSuchElementException.

Our example, each element has 10 seconds.

```
@Test
    public void testImplicitWaits() {
        WebDriver driver = new ChromeDriver();
        // apply implicit wait
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
        driver.get("https://www.saucedemo.com/");
        WebElement usernameInputBox = driver.findElement(By.id("user-name"));
        WebElement passwordInputBox = driver.findElement(By.id("password"));
        WebElement loginButton = driver.findElement(By.id("login-button"));
        usernameInputBox.sendKeys("performance_glitch_user");
        passwordInputBox.sendKeys("secret_sauce");
        loginButton.click();
        WebElement backpackTitle = driver.findElement(By.xpath("//*[text() = 'Sauce Labs
Backpack']"));
        backpackTitle.click();
        WebElement backToHomePage = driver.findElement(By.id("back-to-products"));
        backToHomePage.click();

        WebElement bikeLightTitle = driver.findElement(By.xpath("//*[text() = 'Sauce Labs
Bike Light']"));
        bikeLightTitle.click();

        backToHomePage =driver.findElement(By.id("back-to-products"));
        backToHomePage.click();
        // Close the driver
        driver.quit();
    }
```

Third way : using Explicit Wait

First, let`s see what will happen without using Explicit wait:

***Here we have a case where applying implicit wait will not necessary work.**
*** The problem is that element can sometimes be located but it is not visible yet**
*** In such cases we need to wait and set a condition**

```
*
@Test
    public void testLoadingContent(){
        WebDriver driver = new ChromeDriver();
        driver.get("https://the-internet.herokuapp.com/dynamic_loading");
        driver.findElement(By.xpath(  "//*[contains(text(), 'Example 1:')]")).click();
        WebElement startButton = driver.findElement(By.xpath(  "//*[text() = 'Start']"));
        startButton.click();
        String expectedMessage = "Hello World!";
        WebElement actualFinishMessage = driver.findElement(By.id("finish"));
        Assert.assertEquals (actualFinishMessage.getText(), expectedMessage,  "Finish
message verification failed!");
        driver.quit();

    }
Result:

java.lang.AssertionError: Finish message verification failed!

Expected :Hello World!
Actual   :
```

An **explicit wait** in Selenium WebDriver is a type of wait that allows you to specify a certain condition, and Selenium will pause execution until that condition is met or a maximum amount of time has passed. Unlike implicit waits, which are set globally and apply to all elements, explicit waits are applied only to specific elements or conditions.

Key Features of Explicit Wait

1. **Conditional Waiting**:
    - Explicit waits allow you to wait for specific conditions such as an element becoming visible, clickable, present, or having specific text.
    - This type of wait is more precise compared to implicit waits because it waits for a particular condition rather than just the presence of an element.
2. **Customizable Timeouts**:

    - You can specify the maximum time to wait for the condition to be met.
    - During this time, Selenium will keep polling the condition at regular intervals until either the condition is met or the timeout expires.
3. **Targeted to Specific Elements**:
    - Explicit waits are used for individual elements rather than setting a global wait time for all elements.
    - This makes explicit waits useful when different elements on a page have varying load times or when additional interaction conditions must be met.

Example: Using `WebDriverWait` with Explicit Waits

Explicit waits in Selenium can be implemented using the `WebDriverWait` class along with `ExpectedConditions`. Here's how to use it:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class ExplicitWaitExample {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.saucedemo.com/");

        // Creating an instance of WebDriverWait with a timeout of 10 seconds
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Wait until the username input box is visible before interacting with it
        WebElement usernameInputBox =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("user-name")));
        usernameInputBox.sendKeys("standard_user");

        // Wait until the password input box is visible
        WebElement passwordInputBox =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("password")));
        passwordInputBox.sendKeys("secret_sauce");

        // Wait until the login button is clickable
        WebElement loginButton =
wait.until(ExpectedConditions.elementToBeClickable(By.id("login-button")));
        loginButton.click();

        // Close the driver
        driver.quit();
    }
}
```

Explanation of the Example

1. **`WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));`**
    - This line creates a `WebDriverWait` instance, `wait`, with a timeout of 10 seconds.
    - This means that Selenium will wait up to 10 seconds for a condition to be met.
2. **`wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("user-name")));`**
    - The `wait.until()` method is used to define the condition to be met before proceeding.
    - Here, `ExpectedConditions.visibilityOfElementLocated()` is used to wait until the element with `id="user-name"` becomes visible.
3. **`ExpectedConditions`**
    - The `ExpectedConditions` class provides many common conditions that can be used with explicit waits, such as:
        - **`visibilityOfElementLocated(By locator)`**: Waits for an element to be visible.
        - **`elementToBeClickable(By locator)`**: Waits for an element to be clickable.
        - **`presenceOfElementLocated(By locator)`**: Waits for the element to be present in the DOM.
        - **`textToBePresentInElement(By locator, String text)`**: Waits for specific text to be present in an element.

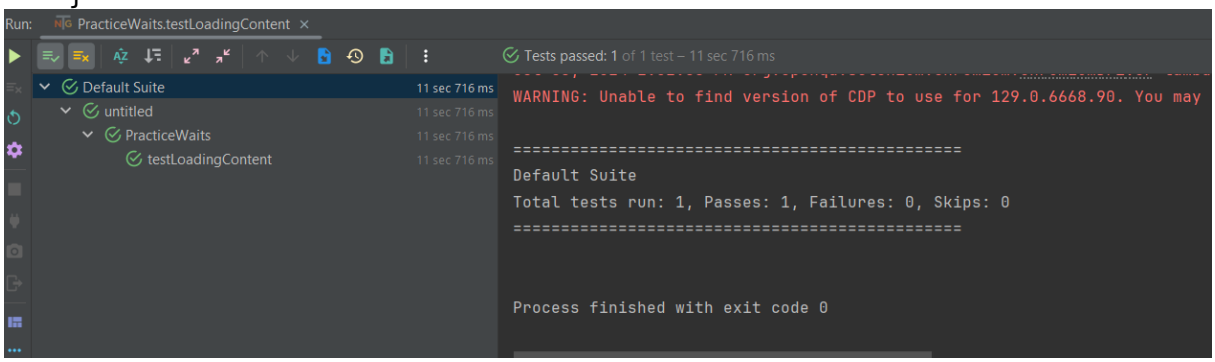
Advantages of Explicit Wait

1. **Precision**:
    - Explicit waits provide more precision by allowing you to wait for a specific condition, making your test more robust and reliable in handling dynamic content.
2. **Reduced Unnecessary Wait Time**:
    - Instead of waiting a fixed duration for all elements, explicit waits only pause when a condition needs to be satisfied, which can lead to more efficient test execution.
3. **Better Error Handling**:
    - Explicit waits allow you to specify a condition and handle a timeout exception if the condition is not met within the given time.

First, let`s see what will happen with using Explicit wait:

```
@Test
    public void testLoadingContent(){
        WebDriver driver = new ChromeDriver();
        driver.get("https://the-internet.herokuapp.com/dynamic_loading");
        driver.findElement(By.xpath(  "//*[contains(text(), 'Example 1:')]")).click();
        WebElement startButton = driver.findElement(By.xpath(  "//*[text() = 'Start']"));
        startButton.click();
        String expectedMessage = "Hello World!";
```

```
        //WebElement actualFinishMessage = driver.findElement(By.id("finish"));
        // First identify the problem line think logically what can be wrong with a
locator
        Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds
(10));
        WebElement actualFinishMessage =
explicitWait.until(ExpectedConditions.visibilityOfElementLocated (By.id("finish")));
        Assert.assertEquals (actualFinishMessage.getText(), expectedMessage,  "Finish
message verification failed!");
        driver.quit();

    }
```


Run:
PracticeWaits.testLoadingContent
Tests passed: 1 of 1 test – 11 sec 716 ms
Default Suite 11 sec 716 ms
untitled 11 sec 716 ms
PracticeWaits 11 sec 716 ms
testLoadingContent 11 sec 716 ms
WARNING: Unable to find version of CDP to use for 129.0.6668.90. You may
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
Process finished with exit code 0

Explaination:

. `driver.findElement(By.xpath("//*[contains(text(), 'Example 1:')]")).click();`

- This line finds and clicks the link with text containing `"Example 1:"`.
- It uses an **XPath locator** to identify an element that contains the specified text.

6. `WebElement startButton = driver.findElement(By.xpath("//*[text() = 'Start']"));`

- This line finds the "Start" button element on the page using **XPath**.
- The `text() = 'Start'` part of the XPath is used to match an element containing the exact text `"Start"`.

7. `startButton.click();`

- This line clicks the **Start** button to trigger the dynamic loading process on the page.

8. `String expectedMessage = "Hello World!";`

- The expected message to verify is stored in the variable `expectedMessage`, which is `"Hello World!"`.

9. Problematic Line Commented Out

```
// WebElement actualFinishMessage = driver.findElement(By.id("finish"));
// First identify the problem line think logically what can be wrong with a locator
```

- This line attempts to locate an element with the ID `"finish"` to retrieve its text.
- It is commented out, likely because locating the element right after clicking the "Start" button would result in a `NoSuchElementException`, as the element may not be immediately present in the DOM.
- Since the content is loaded dynamically, it takes some time to appear, and a direct `findElement()` call will not be sufficient without waiting.

10. Explicit Wait

```
Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement actualFinishMessage =
explicitWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("finish")));
```

- **Explicit Wait** is used to wait until a specific condition is met before proceeding.
- Here, `WebDriverWait` is instantiated with a timeout of 10 seconds, allowing Selenium to keep checking the condition until it is met or the timeout expires.
- `ExpectedConditions.visibilityOfElementLocated(By.id("finish"))` waits for the element with ID `"finish"` to be visible.
- This approach ensures that Selenium will not proceed until the element containing the `"Hello World!"` message is fully loaded and visible.

Using explicit wait for a List:

// Apply explicit wait to a list of web elements

```
List<WebElement> options = explicitWait.until(ExpectedConditions.
visibilityOfAllElementsLocatedBy(By.xpath( xpathExpression: "//")));
```

Explicit Wait Conditions

Mostly used expected conditions:

- **presenceOfElementLocated**(locator)
- **visibilityOfElementLocated**(element)

- **elementToBeClickable**(locator)
- **elementToBeSelected**(locator)

- **textToBePresentInElementLocated**(locator, text)
- **alertIsPresent**()
- **titleIs**(title)

** If any of these conditions are not met before the specified time, then selenium Throws TimoutException.*

- **elementToBeSelected**(locator) → Selenium waits for an element to be selected
  - `wait.until(ExpectedConditions.elementToBeSelected(By.id("tre")));`
- **textToBePresentInElementLocated**() → Selenium waits for an element to have particular text.
  - `wait.until(ExpectedConditions.textToBePresentInElementLocated(By.xpath("//div[@id='timeLeft']"), "Yoll Academy"));`
- **alertIsPresent()** → Selenium waits for an alert to be present
  - `wait.until(ExpectedConditions.alertIsPresent());`
- **titleIs()** → Selenium waits for a webpage to have a particular title
  - `wait.until(ExpectedConditions.titleIs("Bestbuy")));`

Using textToBePresentInElementLocated() Instead of Visibility Check

```
//Option 2 : USING textToBePresentInElementLocated INSTEAD OF : visibilityOfElement
    @Test
    public void testLoadingContenttextToBePresentInElementLocated(){
        WebDriver driver = new ChromeDriver();
        driver.get("https://the-internet.herokuapp.com/dynamic_loading");
        driver.findElement(By.xpath(  "//*[contains(text(), 'Example 1:')]")).click();
        WebElement startButton = driver.findElement(By.xpath(  "//*[text() = 'Start']"));
        startButton.click();
        String expectedMessage = "Hello World!";
// First identify the problem line think logically what can be wrong with a locator
        Wait<WebDriver> explicitWait = new WebDriverWait (driver, Duration.ofSeconds
(10));
```

```
        //USING textToBePresentInElementLocated INSTEAD OF : visibilityOfElement
        //WebElement actual FinishMessage = explicitWait.until
(ExpectedConditions.visibilityOfElement Located (By.id("finish")));
        boolean isTextPresent = explicitWait.until
(ExpectedConditions.textToBePresentInElementLocated (By.id("finish"),  "Hello
Forld!"));
        Assert.assertTrue(isTextPresent, "Text message verification failed!");
    }
    }
```

`ExpectedConditions.textToBePresentInElementLocated(By.id("finish"), "Hello Forld!")`:

- This line sets up a condition to wait for the specific **text** `"Hello Forld!"` to be present in the element identified by `By.id("finish")`.
- If the text appears in the element within 10 seconds, the condition will be met, and `isTextPresent` will be assigned `true`. Otherwise, it will be assigned `false`.
- Using `textToBePresentInElementLocated()` is more reliable in this context than simply checking for the element's visibility, because it ensures the correct text is loaded and available.

Fluent Wait:

```
@Test
    public void testFluent(){
        WebDriver driver = new ChromeDriver();
        driver.get("https://the-internet.herokuapp.com/dynamic_loading");
        driver.findElement(By.xpath(  "//*[contains(text(), 'Example 1:')]")).click();
        WebElement startButton = driver.findElement(By.xpath(  "//*[text() = 'Start']"));
        startButton.click();
        String expectedMessage = "Hello World!";

        Wait<WebDriver> fluentWait = new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds (10))
                .pollingEvery(Duration. ofSeconds(1))
                .ignoring(NoSuchElementException.class)
                .withMessage("The Locator for element failed with polling time of 1 sec
and 10 seconds max limit!");
        WebElement
actualFinishMessage=fluentWait.until(ExpectedConditions.visibilityOfElementLocated
(By.id("finish")));
        Assert.assertEquals(actualFinishMessage.getText(), expectedMessage, "Finish
message verification failed!");
    driver.quit();
    }
```

Pop Ups & Alerts

- Pop-ups are interactive elements that appear over the main content of a webpage
- They are designed to capture user attention for various purposes - from conveying information, gathering inputs, to alerting users about something important.

There are two types of pop-ups:

Web-Based pop-ups:

- Modal pop-ups
- Non-modal pop-ups
- Toast pop-ups

Browser-Based pop-ups:

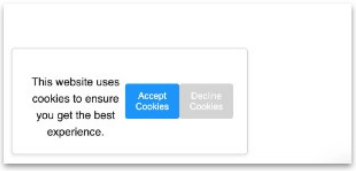
- Alerts

WEB-BASED POP-UPS

Web Based Pop Ups are simply designed **web elements that are a part of html.**

They can **be located in DOM with Selenium strategies** we have learned. Inspecting the DOM, using locators etc…

- **Modal pop-ups drag the full attention from user** by making the main page not interactable. You have to acknowledge first the pop up and then proceed browsing the main page.

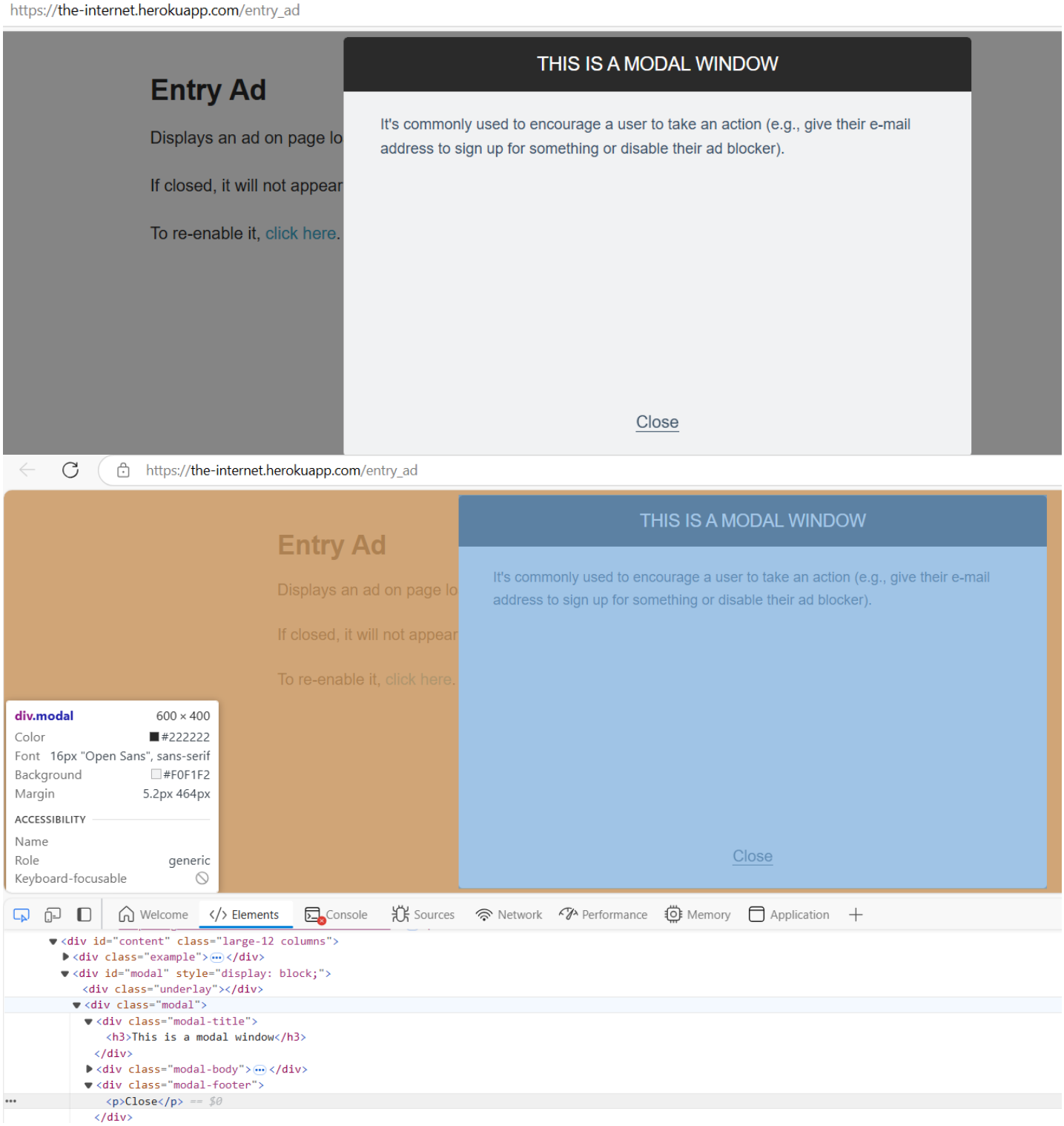


- **Non modal pop-ups show the new window but doesn't interrupt the user interaction** with the main page.
- **Toasts provide gentle notifications to the user** and no additional actions required.

Scenario: Modal pop-ups

Verify user can see a modal pop up once landed on the page
- ○ Given user lands on Entry Ad Page
- ○ When a pop - up appears
- ○ Then user is able to see "THIS IS A MODAL WINDOW"
- ○ And user can close the pop up

Web Site Url: the-internet.herokuapp.com

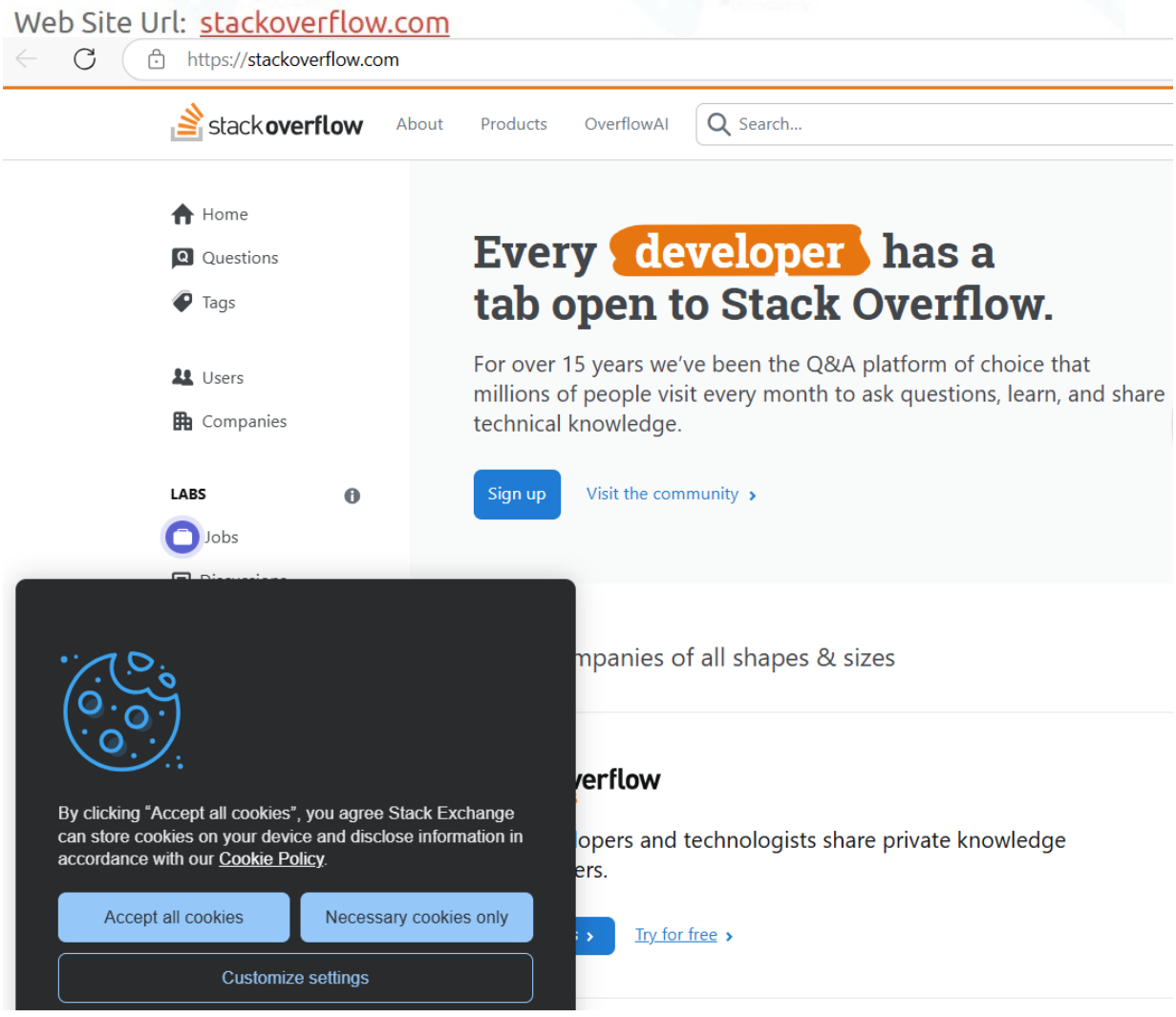



```
package practice;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
```

```java
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.Wait;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.time.Duration;


public class HerokuAppTest {
    WebDriver driver;
    @BeforeMethod
    public void setUp(){

        driver= new ChromeDriver();
        driver.get("https://the-internet.herokuapp.com/entry_ad");

    }

    @AfterMethod
    public void tearDown(){
        driver.quit();}
    @Test
    public void testUserCanSeePopUpOnceLandedOnTheHomePage() throws InterruptedException
{

        // Apply explicit waits to wait for the pop-up to appear properly
        Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds(10));
        // Wait for the pop-up modal to be visible
        By popUpModal=By.cssSelector("div.modal");
        explicitWait.until(ExpectedConditions.visibilityOfElementLocated(popUpModal));
        // Verify the title of the pop-up
        WebElement popUpTitle = driver.findElement(By.cssSelector("div.modal-title"));
        String expectedPopUpTitle = "THIS IS A MODAL WINDOW";
        Assert.assertEquals(popUpTitle.getText(), expectedPopUpTitle, "Pop Up Title
verification Failed");
        // Close the pop-up by clicking the close button
        WebElement closePopUpButton = driver.findElement(By.cssSelector(".modal-
footer>p"));
        closePopUpButton.click();
        // Wait for the pop-up to become Displayed after clicking the close button
        boolean isPopUpDisplayed = driver.findElement(popUpModal).isDisplayed();
        // Assert that the pop-up is no longer Displayed
        Assert.assertFalse(isPopUpDisplayed,  "Pop Up is still visible!");
    }
}
```

Scenario: Modal pop-ups

- **Scenario:** Verify user can see a non-modal pop-up once landed on the page
  - ***Given*** user lands on stackoverflow app
  - ***When*** the cookies pop up appears
  - ***Then*** user is able to accept cookies
  - ***And*** the pop up disappears


Web Site Url: stackoverflow.com
https://stackoverflow.com
stackoverflow
About
Products
OverflowAI
Search...
Home
Questions
Tags
Users
Companies
LABS
Jobs
Every developer has a tab open to Stack Overflow.
For over 15 years we've been the Q&A platform of choice that millions of people visit every month to ask questions, learn, and share technical knowledge.
Sign up
Visit the community
By clicking "Accept all cookies", you agree Stack Exchange can store cookies on your device and disclose information in accordance with our Cookie Policy.
Accept all cookies
Necessary cookies only
Customize settings
Try for free

```
package practice;
```

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.Wait;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.time.Duration;

public class StackOverFlowTest {
    WebDriver driver;
    @BeforeMethod
    public void setUp(){

        driver= new ChromeDriver();
        driver.get("https://stackoverflow.com/");
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
    }

    @AfterMethod
    public void tearDown(){
        driver.quit();}


    @Test
    public void userCanHandleACookiesPopUp() throws InterruptedException {
        Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds(10));
        WebElement necessaryCookiesButton =
explicitWait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(
"//*[contains(text(), 'Necessary cookies only')]")));
        necessaryCookiesButton.click();
        boolean isPopUpInvisible
=explicitWait.until(ExpectedConditions.invisibilityOfElementLocated(By.xpath(
"//*[contains(text(), 'Necessary cookies only')]")));
        Assert.assertTrue (isPopUpInvisible,  "Pop Up verification failed.");
}}
```

Explaination

Step 1: Explicit Wait Setup

```
Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds(10));
```

- **Explicit Wait**: Here, an explicit wait is created with a timeout of 10 seconds using `WebDriverWait`. An explicit wait allows the test to wait for specific conditions (like visibility of an element) before proceeding, making it useful when dealing with dynamic content like pop-ups that may not be immediately available.

- The **`Duration.ofSeconds(10)`** specifies the maximum time that Selenium should wait for the condition to be met before throwing a `TimeoutException`.

Step 2: Waiting for the "Necessary Cookies Only" Button

```
WebElement necessaryCookiesButton =
explicitWait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//
*[contains(text(), 'Necessary cookies only')]")));
```

- **Find the Button Using XPath**: This line waits for the "Necessary cookies only" button to become visible on the page. The `ExpectedConditions.visibilityOfElementLocated()` method waits until the element located by the given XPath (which contains the text "Necessary cookies only") is visible in the DOM.
- **`WebElement necessaryCookiesButton`**: Once the element becomes visible, it is assigned to the `necessaryCookiesButton` WebElement, allowing you to interact with it.

Step 3: Clicking the Button

```
necessaryCookiesButton.click();
```

- This line clicks on the "Necessary cookies only" button, which dismisses the cookies pop-up.

Step 4: Waiting for the Pop-up to Disappear

```
boolean isPopUpInvisible =
explicitWait.until(ExpectedConditions.invisibilityOfElementLocated(By.xpath("//
*[contains(text(), 'Necessary cookies only')]")));
```

- **Waiting for Pop-up to Disappear**: After clicking the button, this line waits for the pop-up (specifically the "Necessary cookies only" button) to become invisible. The `ExpectedConditions.invisibilityOfElementLocated()` method returns `true` if the element is no longer visible.
- **`isPopUpInvisible`**: This boolean stores the result of waiting for the element's invisibility. If the element becomes invisible, the value will be `true`.

Step 5: Assertion

```
Assert.assertTrue(isPopUpInvisible, "Pop Up verification failed.");
```

- **`Assert.assertTrue()`**: This asserts that the `isPopUpInvisible` variable is `true`, meaning the pop-up has disappeared. If `isPopUpInvisible` is `false`, the assertion will fail, and the test will report "Pop Up verification failed."

- **Assertion Logic**: The test expects that after clicking the button, the pop-up will no longer be visible, so the assertion checks for this condition. If the pop-up is still visible, the test fails.

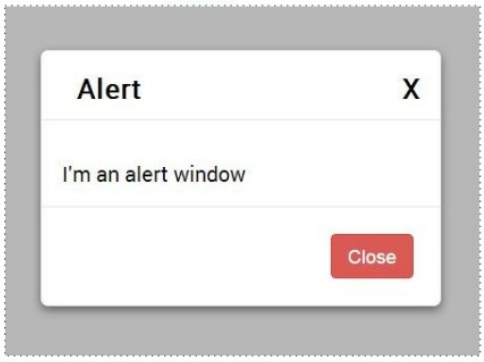
Summary of Test Flow:

1. **Wait for the cookies pop-up button**: Waits until the "Necessary cookies only" button is visible.
2. **Click the button**: Clicks the button to dismiss the cookies pop-up.
3. **Wait for the pop-up to disappear**: Verifies that the pop-up becomes invisible after the button is clicked.
4. **Assertion**: The test asserts that the pop-up is no longer visible after the button is clicked.

This test ensures the pop-up is correctly handled on the webpage, making it a reliable way to interact with dynamic content that may load after the page itself.

Browser based Pop-ups (Alerts)

- Alerts freeze the current window, takes the focus away from the current window and forces the browser to read the message.
- Alert shapes/designs are different on each browser.
- There are 3 types of alerts
  - Alert
  - Confirmation Dialog
  - Prompt



ALERTS

- Alerts have certain characteristics:
  - We cannot locate alerts using inspect tools (id, name, css, xpath etc..)
  - Alerts are not html windows
- When using Selenium, if alert shows up as part of the automation, we have to handle that alert before proceeding.
- Otherwise we get: UnhandledAlertException, and test exits.
- We can use Explicit wait / WebdriverWait to check whether the alert is there or not.
  - `wait.until(ExpectedConditions.alertIsPresent());`

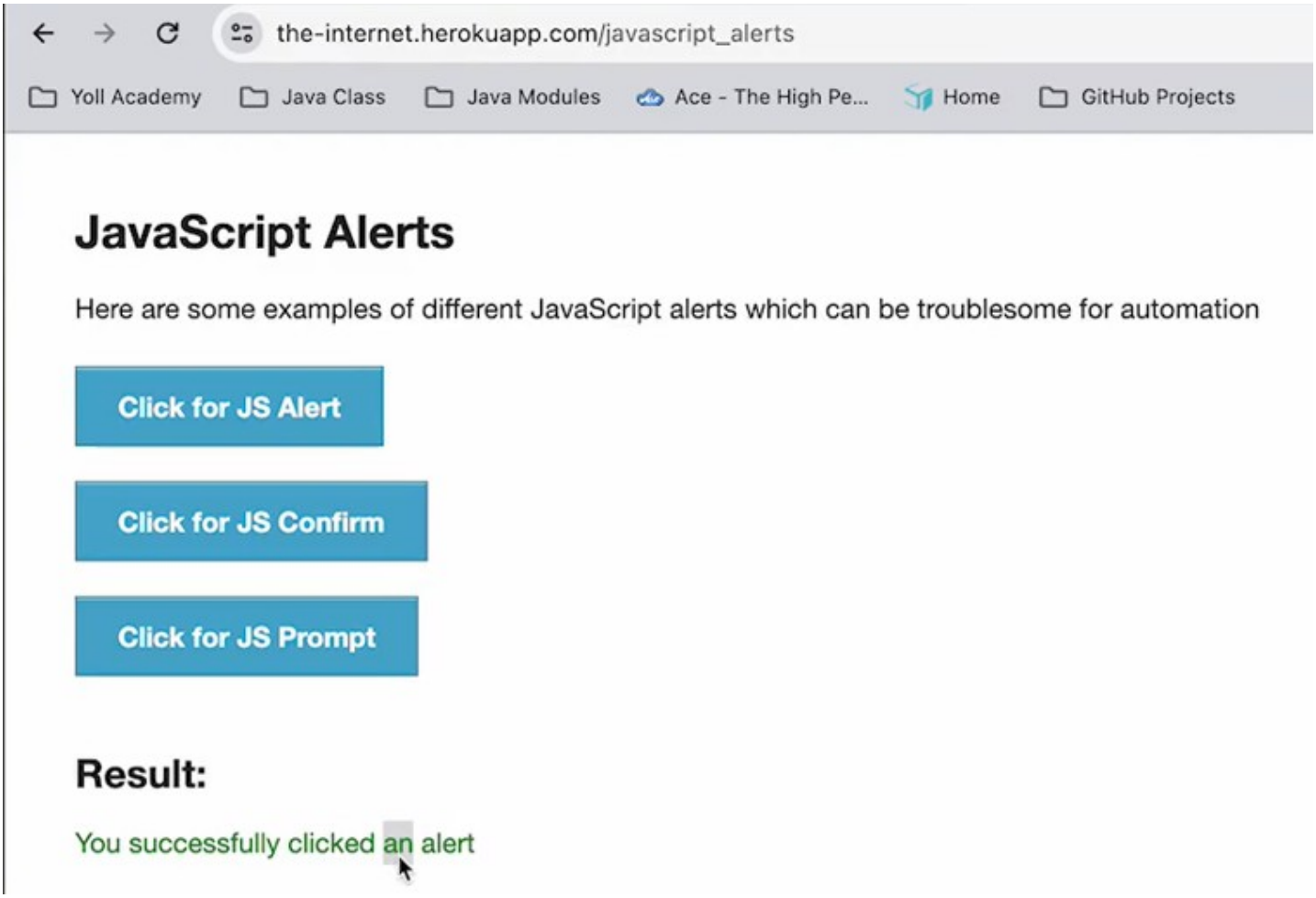
HANDLING ALERTS

- We handle alerts using:
  - `Alert myAlert = driver.switchTo().alert();`
  - By calling switchTo() method, we are passing control of selenium from html page to alert popup
  - There are 4 actions we can make with alerts
    - `myAlert.accept();` → OK
    - `myAlert.dismiss();` → Cancel
    - `myAlert.getText();` → Get alert text
    - `myAlert.sendKeys("test Text");` → Type into the prompt

Scenario: Navigate to https://testautomationpractice.blogspot.com/

●Click 'click me' for alert practice
●Print alert text

- **Accept the alert**
- **Click ‘click me’ for alert one more time**
- **Dismiss the alert**
- **Print text that appeared in UI**




```
@Test
    public void testSuccessfulAcceptOfSimpleAlert() throws InterruptedException {
        driver.get("https://the-internet.herokuapp.com/javascript_alerts");
        WebElement clickForJsAlertButton = driver.findElement(By.xpath("//*[text() =
'Click for JS Alert']"));
        clickForJsAlertButton.click();

Wait<WebDriver> explicitWait = new WebDriverWait (driver, Duration.ofSeconds (10));
explicitWait.until (ExpectedConditions. alertIsPresent());


        Alert alert = driver.switchTo().alert();
        alert.accept();
        String expectedMessage = "You successfully clicked an alert";
        WebElement resultMessage = driver.findElement(By.id("result"));
```

```
        Assert.assertEquals(resultMessage.getText(), expectedMessage, "Message
verification after alert handling failed!");
    }
```

Explaination:

Purpose of the Test:

This test verifies that the user can successfully accept a simple JavaScript alert and check that the correct message is displayed after the alert is handled.

Step 1: Navigate to the Target Web Page

```
java
Copy code
driver.get("https://the-internet.herokuapp.com/javascript_alerts");
```

- **`driver.get()`**: Navigates to the page that contains JavaScript alert examples.

Step 2: Locate the "Click for JS Alert" Button

```
WebElement clickForJsAlertButton = driver.findElement(By.xpath("//*[text() = 'Click for
JS Alert']"));
clickForJsAlertButton.click();
```

- **`driver.findElement()`**: Finds the button on the page that triggers a simple JavaScript alert by using its visible text, "Click for JS Alert", through an XPath expression.
- **Button Click**: Once the button is located, the `click()` method is called to trigger the JavaScript alert.

Step 3: Wait for the Alert to Appear

```
Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds(10));
explicitWait.until(ExpectedConditions.alertIsPresent());
```

- **Explicit Wait Setup**: An explicit wait is created with a timeout of 10 seconds. This will pause the execution until a condition is met or the timeout expires.
- **`ExpectedConditions.alertIsPresent()`**: This condition waits until a JavaScript alert is present on the page. The test will not proceed until the alert has appeared or the 10-second timeout has passed.

Step 4: Handle the Alert

```
Alert alert = driver.switchTo().alert();
```

```
alert.accept();
```

- **`driver.switchTo().alert()`**: This switches the driver's context to the alert that has appeared. The `Alert` interface is used to interact with browser alerts.
- **`alert.accept()`**: This method accepts (or confirms) the alert, which is equivalent to clicking the "OK" button on the alert.

Step 5: Verifying the Result Message

```
String expectedMessage = "You successfully clicked an alert";
WebElement resultMessage = driver.findElement(By.id("result"));
Assert.assertEquals(resultMessage.getText(), expectedMessage, "Message verification after alert handling failed!");
```

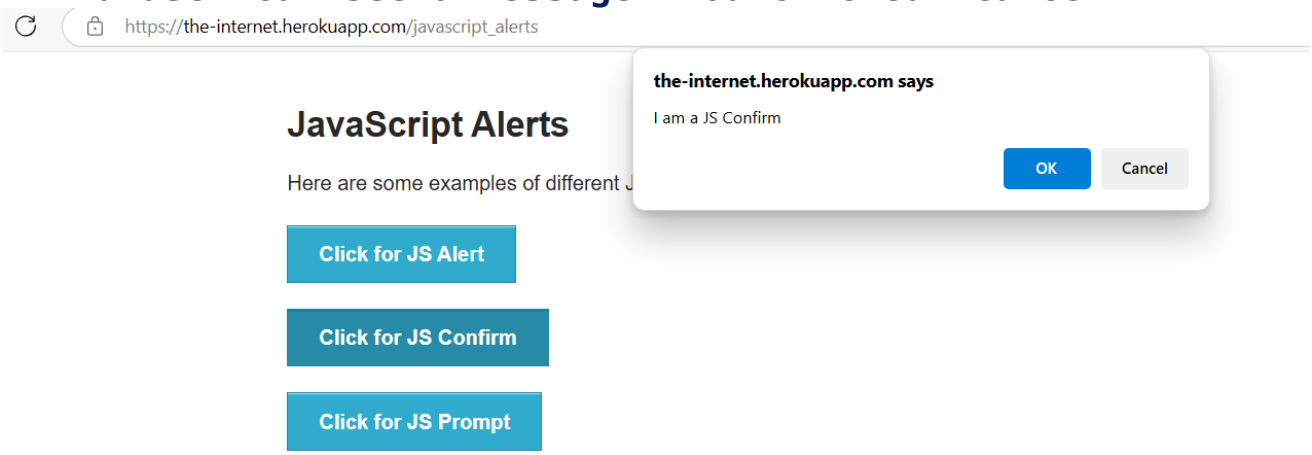
- **Expected Message**: The variable `expectedMessage` holds the expected text that should appear after the alert is accepted.
- **Locate Result Message**: `driver.findElement(By.id("result"))` finds the result message on the page. This is the text that will appear on the page after interacting with the alert.
- **`Assert.assertEquals()`**: This assertion compares the actual message from the page (`resultMessage.getText()`) with the expected message (`"You successfully clicked an alert"`). If they do not match, the test will fail, and the error message "Message verification after alert handling failed!" will be displayed.

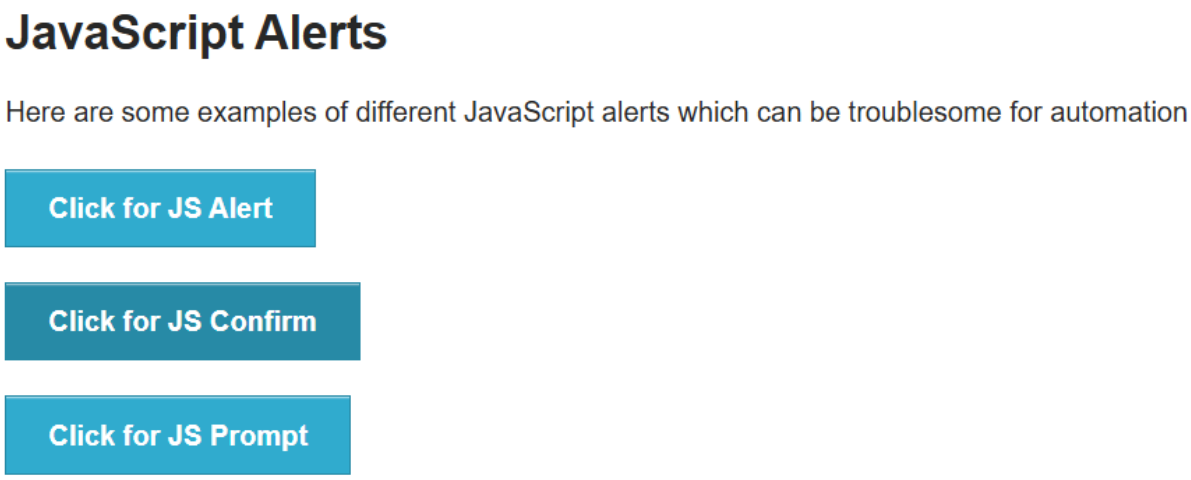
Summary of the Test Flow:

1. **Navigate to the target page**: The test navigates to the page with JavaScript alerts.
2. **Click the "Click for JS Alert" button**: This button triggers a simple JavaScript alert.
3. **Wait for the alert to appear**: The test waits for the alert to be present using an explicit wait.
4. **Handle the alert**: The test accepts the alert by clicking the "OK" button.
5. **Verify the result**: The test checks if the correct message is displayed on the page after the alert is handled, ensuring that the alert interaction was successful.

Scenario: Verify user can accept a confirm alert
Given user lands on https://the-internet.herokuapp.com/javascript_alerts
When user clicks on Click for JS Confirm

And an alert appearsThen user can see the alert message is "I am a JS Confirm"
And user is able to decline the alert
And user can see a message "You clicked: Cancel"




JavaScript Alerts
Here are some examples of different JavaScript alerts which can be troublesome for automation
Click for JS Alert
Click for JS Confirm
Click for JS Prompt
Result:
You clicked: Ok

```
@Test
    public void testUserCanDismissAConfirmAlert() {
        driver.get("https://the-internet.herokuapp.com/javascript_alerts");
        WebElement clickForJsConfirm = driver.findElement(By.xpath(  "//*[text() = 'Click
for JS Confirm']"));
        clickForJsConfirm.click();
```

```
        String expectedAlertMessage = "I am a JS Confirm";
        Wait<WebDriver> explicitWait = new WebDriverWait (driver, Duration.ofSeconds
(10));
        explicitWait.until (ExpectedConditions. alertIsPresent());
        Alert alert = driver.switchTo().alert();
        String actualAlertMessage = alert.getText();
        Assert.assertEquals(actualAlertMessage, expectedAlertMessage,  "Alert message
verification failed!");
        alert.dismiss();
        String expectedMessageAfterAlertHandled = "You clicked: Cancel";
        WebElement resultMessage = driver.findElement(By.id("result"));
        Assert.assertEquals
(resultMessage.getText(),expectedMessageAfterAlertHandled,"Message verification after
bnbj handling failed!");

}
```

Explaination:

Purpose of the Test:

The test verifies that:

1. The correct JavaScript confirmation alert is displayed with the expected message.
2. The alert can be dismissed (by clicking "Cancel").
3. The correct result message is displayed after dismissing the alert.

Code Breakdown:

```
public void testUserCanDismissAConfirmAlert() {
```

- **Method Name**: The name `testUserCanDismissAConfirmAlert` suggests the purpose of the test: to ensure that the user can interact with and dismiss a JavaScript confirmation alert.

Step 1: Navigate to the Target Web Page

```
driver.get("https://the-internet.herokuapp.com/javascript_alerts");
```

- **`driver.get()`**: Navigates the browser to the URL that contains JavaScript alert examples. In this case, the page provides several examples of JavaScript alerts, including confirmation alerts.

Step 2: Locate and Click the "Click for JS Confirm" Button

```
WebElement clickForJsConfirm = driver.findElement(By.xpath("//*[text() = 'Click for JS
Confirm']"));
clickForJsConfirm.click();
```

- **`driver.findElement()`**: Finds the "Click for JS Confirm" button using an XPath locator that looks for the exact text.
- **Button Click**: Once the button is located, the `click()` method is called, which triggers the confirmation alert.

Step 3: Waiting for the Alert to Appear and Verifying its Message

```
String expectedAlertMessage = "I am a JS Confirm";
Wait<WebDriver> explicitWait = new WebDriverWait(driver, Duration.ofSeconds(10));
explicitWait.until(ExpectedConditions.alertIsPresent());
```

- **Expected Alert Message**: The variable `expectedAlertMessage` holds the message you expect to see inside the confirmation alert.
- **Explicit Wait Setup**: A wait with a timeout of 10 seconds is set up to pause execution until the alert is present. The `ExpectedConditions.alertIsPresent()` waits for the JavaScript alert to appear.

```
Alert alert = driver.switchTo().alert();
String actualAlertMessage = alert.getText();
Assert.assertEquals(actualAlertMessage, expectedAlertMessage, "Alert message verification
failed!");
```

- **Switch to Alert**: `driver.switchTo().alert()` switches the WebDriver context to the alert so that interactions with the alert are possible.
- **Get Alert Text**: `alert.getText()` retrieves the actual message displayed inside the alert.
- **`Assert.assertEquals()`**: This assertion checks if the actual alert message matches the expected message (`"I am a JS Confirm"`). If the messages don't match, the test will fail, displaying "Alert message verification failed!".

Step 4: Dismissing the Alert (Click "Cancel")

```
alert.dismiss();
```

- **Dismiss the Alert**: The `alert.dismiss()` method clicks the "Cancel" button on the confirmation alert. In the context of a confirmation alert, "dismiss" simulates the action of clicking "Cancel", while `alert.accept()` would simulate clicking "OK."

Step 5: Verifying the Result Message After Dismissing the Alert

```
String expectedMessageAfterAlertHandled = "You clicked: Cancel";
WebElement resultMessage = driver.findElement(By.id("result"));
Assert.assertEquals(resultMessage.getText(), expectedMessageAfterAlertHandled, "Message
verification after alert handling failed!");
```

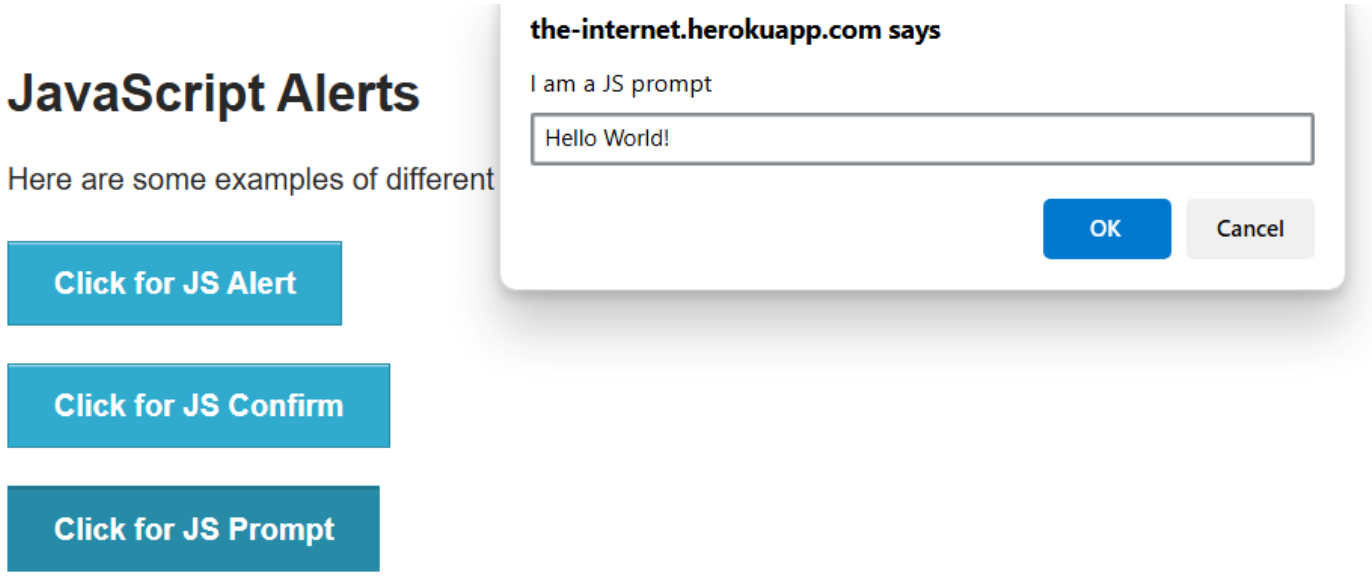
- **Expected Message After Dismissal**: The variable `expectedMessageAfterAlertHandled` holds the text that should be displayed after the alert is dismissed (i.e., after clicking "Cancel"). In this case, the expected text is `"You clicked: Cancel"`.
- **Locate the Result Message**: The message that appears on the webpage after handling the alert is located by its ID (`"result"`).
- **`Assert.assertEquals()`**: This assertion checks whether the actual result message matches the expected message. If the text does not match, the test will fail with the error message "Message verification after alert handling failed!"

Summary of the Test Flow:

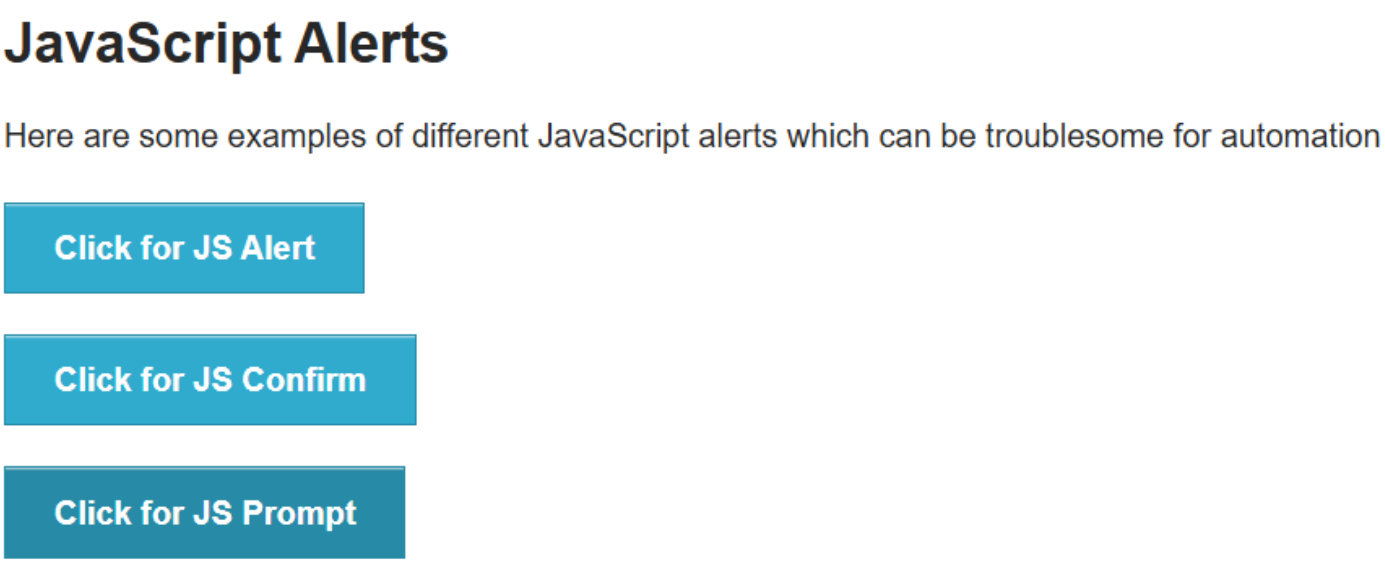
1. **Navigate to the page**: The test navigates to a webpage that demonstrates JavaScript alerts.
2. **Click the "Click for JS Confirm" button**: This button triggers a JavaScript confirmation alert.
3. **Wait for the alert**: The test waits for the confirmation alert to appear using an explicit wait.
4. **Verify the alert message**: The test checks if the message inside the alert matches the expected message.
5. **Dismiss the alert**: The test clicks the "Cancel" button on the confirmation alert.
6. **Verify the result message**: The test verifies that the correct message (`"You clicked: Cancel"`) is displayed on the webpage after dismissing the alert.

Scenario: Verify user can input text in a prompt alert
Given user lands on
https://the-innternet.herokuapp.com/javascript_alerts

When user clicks on Click for JS Prompt
And an alert appears
Then user can see the alert message is "I am a JS prompt"
And user is able to input text "Hello World!" into prompt alert
And user is able to accept the alert
And user can see a message "You entered: Hello World!"



Result:


JavaScript Alerts
Here are some examples of different JavaScript alerts which can be troublesome for automation
Click for JS Alert
Click for JS Confirm
Click for JS Prompt

Result:

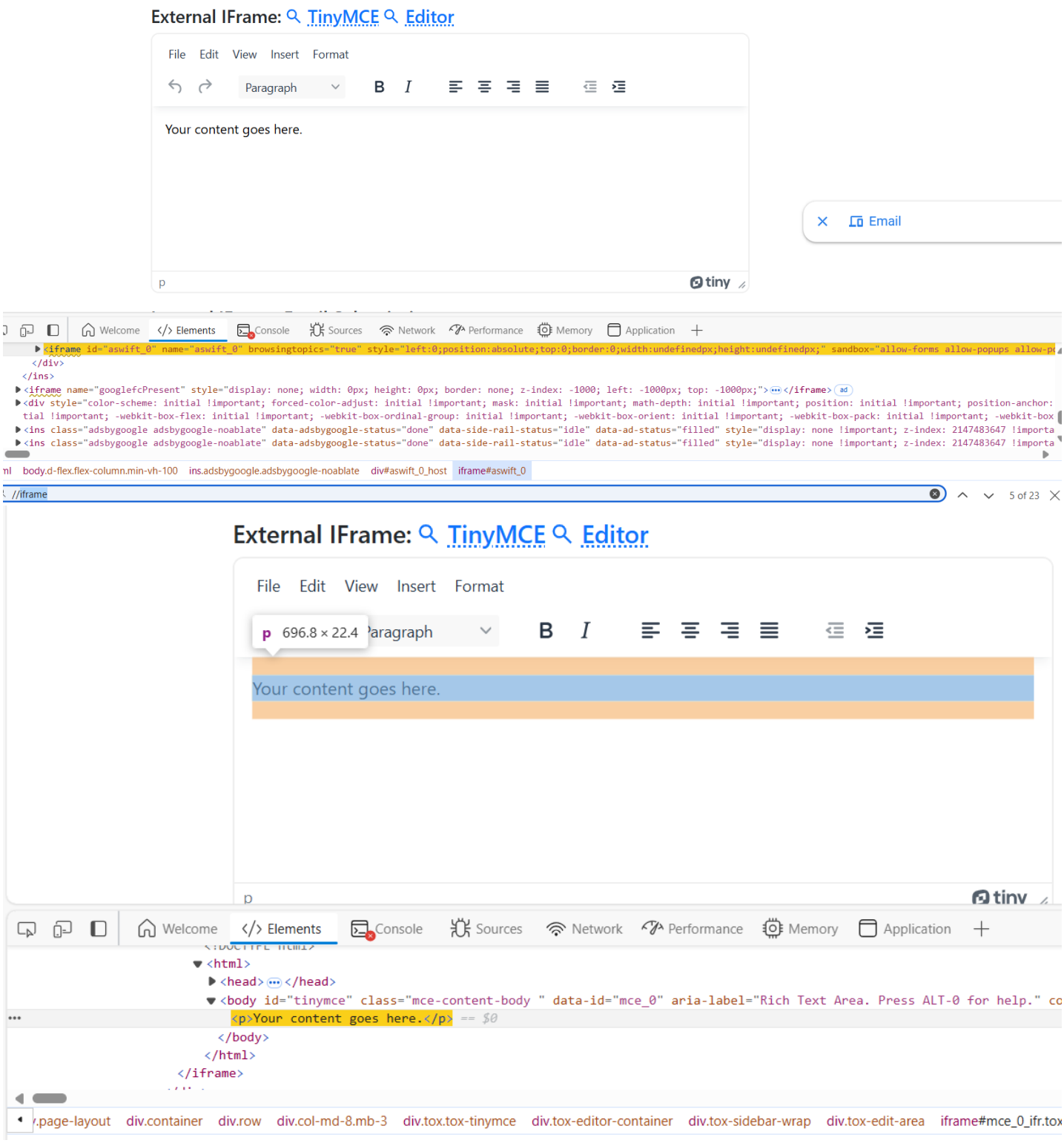
You entered: Hello World!

```
@Test
    public void testUserCanInputDataInPromptAlert() {
        driver.get("https://the-internet.herokuapp.com/javascript_alerts");
        WebElement clickForJsConfirm = driver.findElement(By.xpath(  "//*[text() = 'Click
for JS Prompt']"));
```

```
        clickForJsConfirm.click();

        Wait<WebDriver> explicitWait = new WebDriverWait (driver, Duration.ofSeconds
(10));
        explicitWait.until (ExpectedConditions. alertIsPresent());
        Alert alert = driver.switchTo().alert();
        String expectedAlertMessage = "I am a JS prompt";

        String actualAlertMessage = alert.getText();
        Assert.assertEquals(actualAlertMessage, expectedAlertMessage,  "Alert message
verification failed!");
        String textToSend="Hello World!";
        alert.sendKeys(textToSend);
        alert.accept();

        String expectedMessageAfterAlertHandled = "You entered: Hello World!";
        WebElement resultMessage = driver.findElement(By.id("result"));
        Assert.assertEquals
(resultMessage.getText(),expectedMessageAfterAlertHandled,"Message verification after
handling failed!");

    }
```

Hint:
Use //iframe to see the number of frames on the html code

Scenario: Verify user can interact with text input box and format the text
Given user lands to http://the-internet.herokuapp.com/
When user clicks on “Frames” tab
And user clicks on “iFrames” tab
Then user can type something into text box
And user can format the text



First Way Wrong Aspect without switching too the Iframe

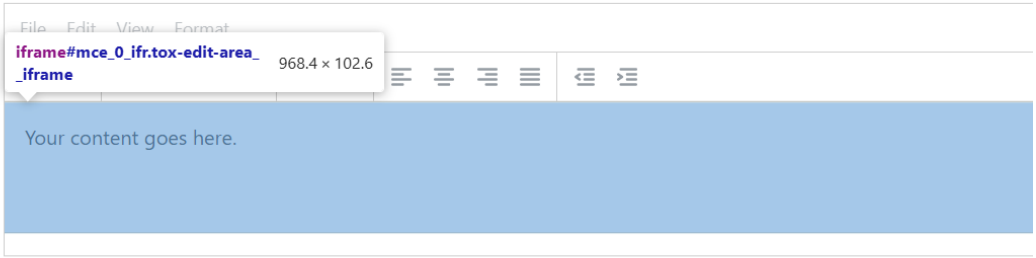
```
package practice;
```

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.time.Duration;

public class FrameTest {
    WebDriver driver;

    @BeforeMethod
    public void setUp() {

        driver = new ChromeDriver();
        //driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(5));
    }

    @AfterMethod
    public void tearDown() {
        driver.quit();
    }
    @Test
    public void herokuAppFramesPractice() throws InterruptedException {
        driver.get("https://the-internet.herokuapp.com/");

//When user clicks on "Frames" tab
        WebElement framesTab = driver.findElement(By.xpath(  "//*[text() = 'Frames']"));
framesTab.click();
        Thread.sleep(5000);
//And user clicks on "iFrames" tab
        WebElement iFameTab = driver.findElement(By.xpath(  "//*[text() = 'iFrame']"));
        iFameTab.click();
        Thread.sleep(5000);
//Then user can type something into text box
        WebElement textBox = driver.findElement(By.cssSelector("#tinymce>p"));
        textBox.sendKeys( "Hello, I'm SDET!");
        Thread.sleep(5000);
//And user can format the text
    }

}
```

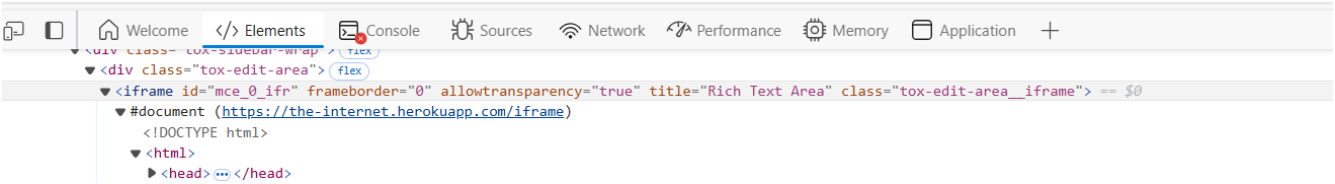
Second Way with switching too the Iframe

Iframe

An iFrame containing the TinyMCE WYSIWYG Editor

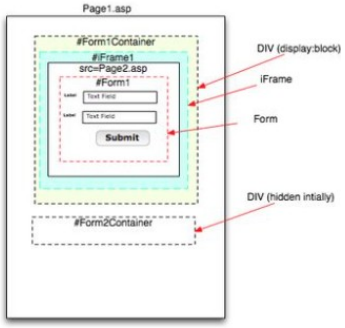


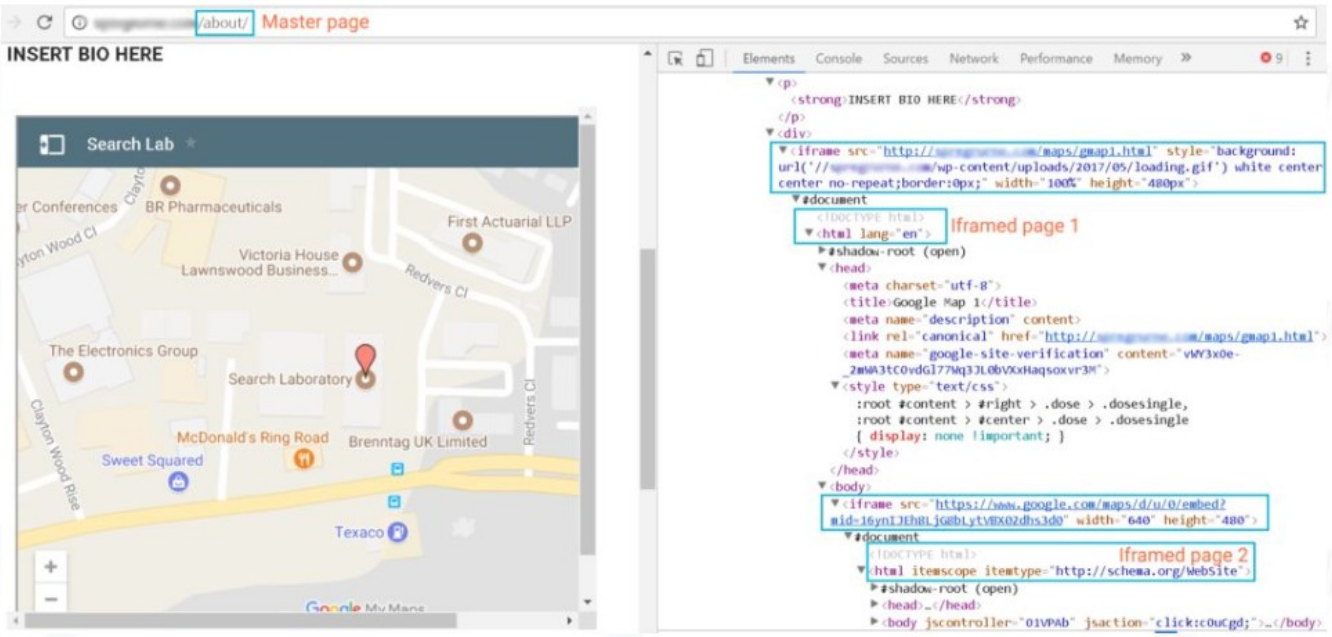
Powered by Elemental Selenium



FRAMES & IFRAMES

- **Frame** → is an **HTML tag** and used to divide the same web page into various frames/windows. Some content is displayed in those frames.
- **iFrame** → (inline frame) is similar to frame, except it can embed other web pages inside the same html.
  - Example: Ads embedded in the webpage, maps etc.
- As far as selenium is concerned, both Frame and iFrame are handled same way.





Switching to IFrames

- The reason we need to be aware of frames/iframes is because selenium **cannot interact** with an element directly if it is inside them.

- There is additional step needs to be done before we can interact with those elements.
- That step is called, **switching the control** from the main html page to the iframe
- There are 3 ways to switch the control

    - By **index** → `driver.switchTo().frame(0)`
        - Since the order of frames can change, it is not reliable way

    - By **name** or **ID** → `driver.switchTo().frame("customIframe");`
        - If these attributes are not present can't use it.

    - By **webelement** →
        - `WebElement testFrame = driverfindElement(By.xpath("//*[@class='tls']"));`
        - `driver.switchTo().frame(testFrame)`
        - If no unique ID or name is present, then we use xpath, css etc. to locate the iframe element

Need to be corrected

```
package practice;

import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
```

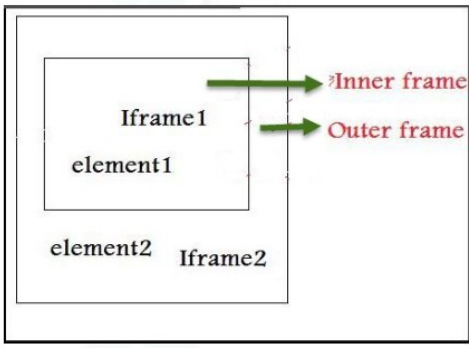
```java
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.time.Duration;

public class FrameTest {
    WebDriver driver;

    @BeforeMethod
    public void setUp() {

        driver = new ChromeDriver();
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(30));
    }

    @AfterMethod
    public void tearDown() {
        driver.quit();
    }
    @Test
    public void herokuAppFramesPractice() throws InterruptedException {
        driver.get("https://the-internet.herokuapp.com/");

//When user clicks on "Frames" tab
        WebElement framesTab = driver.findElement(By.xpath(  "//*[text() = 'Frames']"));
        framesTab.click();
        Thread.sleep(10000);
//And user clicks on "iFrames" tab
        WebElement iFameTab = driver.findElement(By.xpath(  "//*[text() = 'iFrame']"));
        iFameTab.click();
        Thread.sleep(10000);
        //driver.switchTo().frame("mce-0-ifr");
        driver.switchTo().frame("mce_0_ifr");


//Then user can type something into text box
        WebElement textBox = driver.findElement(By.cssSelector("#tinymce>p"));
        textBox.clear();
        textBox.sendKeys( "Hello, I'm SDET!");
        Thread.sleep(20000);

    // Highlight all text Ctrl+ a or Command + a
        textBox.sendKeys(Keys.CONTROL + "a");
    //And user can format the text
    // before interacting with main page elements we have to switch back to it
        driver.switchTo().defaultContent();
        WebElement boldButton = driver.findElement(By.cssSelector("[title= 'Bold']"));
        boldButton.click();
        Thread.sleep( 5000);
//And user can format the text
    }
```

}

Nested Iframes

- There are times where one iframe is located inside another.
- These are called nested iframes

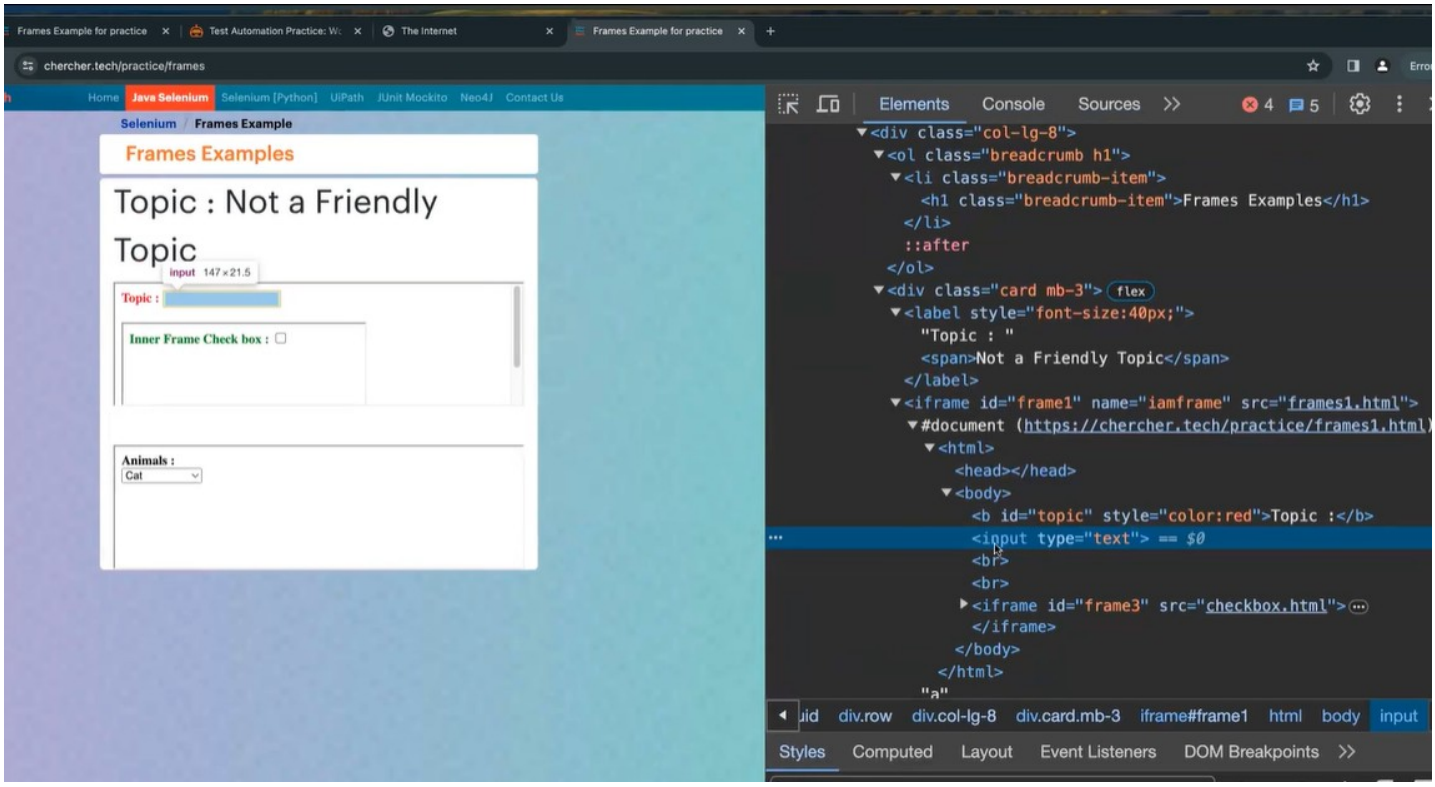
- We cannot directly switch to inner iframe.
- We need to switch to outer iframe first, and then switch to inner iframe.
  - `driver.switchTo().frame("iframe2");`
  - `driver.switchTo().frame("iframe1");`
- You cannot also go between child iframes directly. Have to go up to parent and then come down to sibling.



- To switch control back to parent iframe
  - `driver.switchTo().parentFrame();`

- To switch control back to **main html page**
  - `driver.switchTo().defaultContent();`

Scenario: Verify user can interact with web elements on the displayedGiven user navigates to https://chercher.tech/practice/frames > does not work
Then user can see the page header “Frames Example”
And user can see “Topic : Not a Friendly Topic”
And user is able to type into Topic input box
And user is able to check the “Inner Frame Check box”
And user is able to select “Avatar” from Animals dropdown

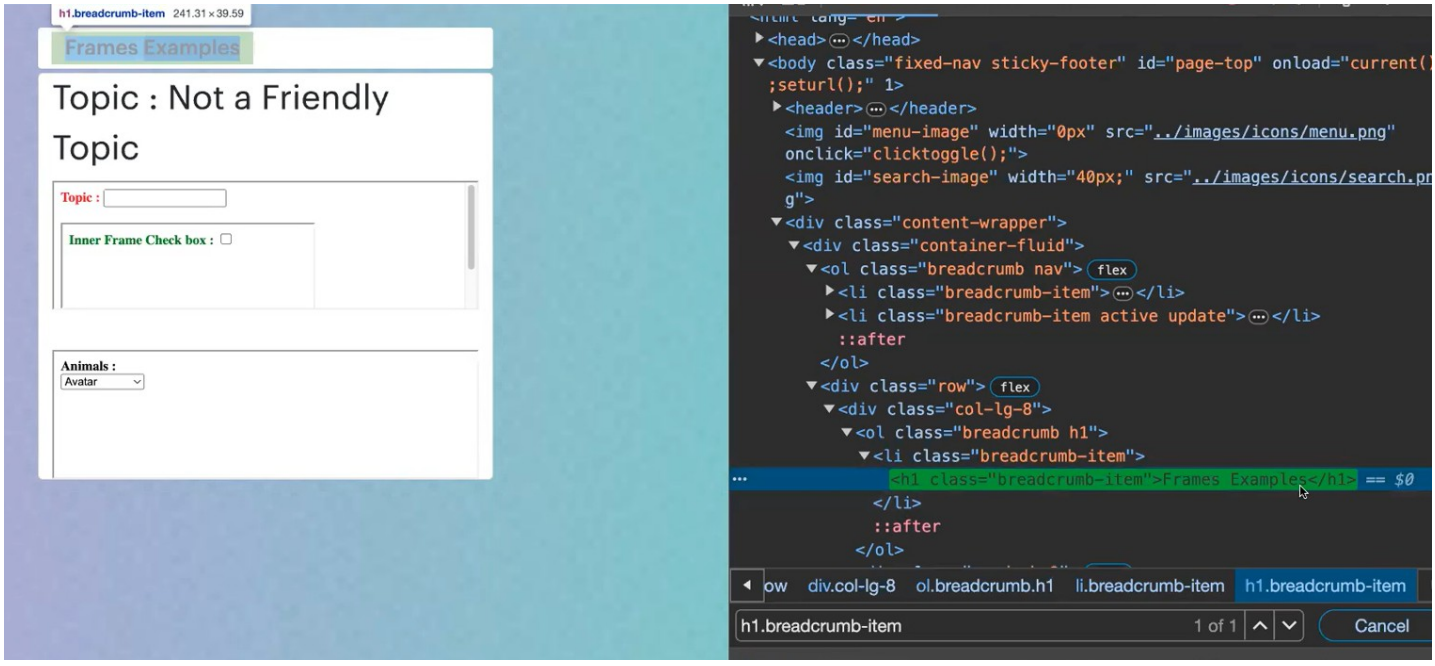

Frames Example for practice
Test Automation Practice: W
The Internet
Frames Example for practice
chercher.tech/practice/frames
Home
Java Selenium
Selenium [Python]
UiPath
JUnit Mockito
Neo4J
Contact Us
Selenium / Frames Example
Frames Examples
Topic : Not a Friendly Topic
input 147 × 21.5
Topic :
Inner Frame Check box :
Animals :
Cat
Elements
Console
Sources

<ol class="breadcrumb h1">
<li class="breadcrumb-item">
<h1 class="breadcrumb-item">Frames Examples</h1>

::after

flex
<label style="font-size:40px;">
"Topic : "
Not a Friendly Topic
</label>
<iframe id="frame1" name="iamframe" src="frames1.html">
#document (https://chercher.tech/practice/frames1.html)
<html>
<head></head>
<body>
Topic :
<input type="text"> == $0

<iframe id="frame3" src="checkbox.html">
</iframe>
</body>
</html>
div.row
div.col-lg-8
div.card.mb-3
iframe#frame1
html
body
input
Styles
Computed
Layout
Event Listeners
DOM Breakpoints


h1.breadcrumb-item 241.31 × 39.59
Frames Examples
Topic : Not a Friendly Topic
Topic :
Inner Frame Check box :
Animals :
Avatar
<head></head>
<body class="fixed-nav sticky-footer" id="page-top" onload="current()
;seturl();" 1>
<header></header>
<img id="menu-image" width="0px" src="../images/icons/menu.png"
onclick="clicktoggle();">
<img id="search-image" width="40px;" src="../images/icons/search.pn
g">

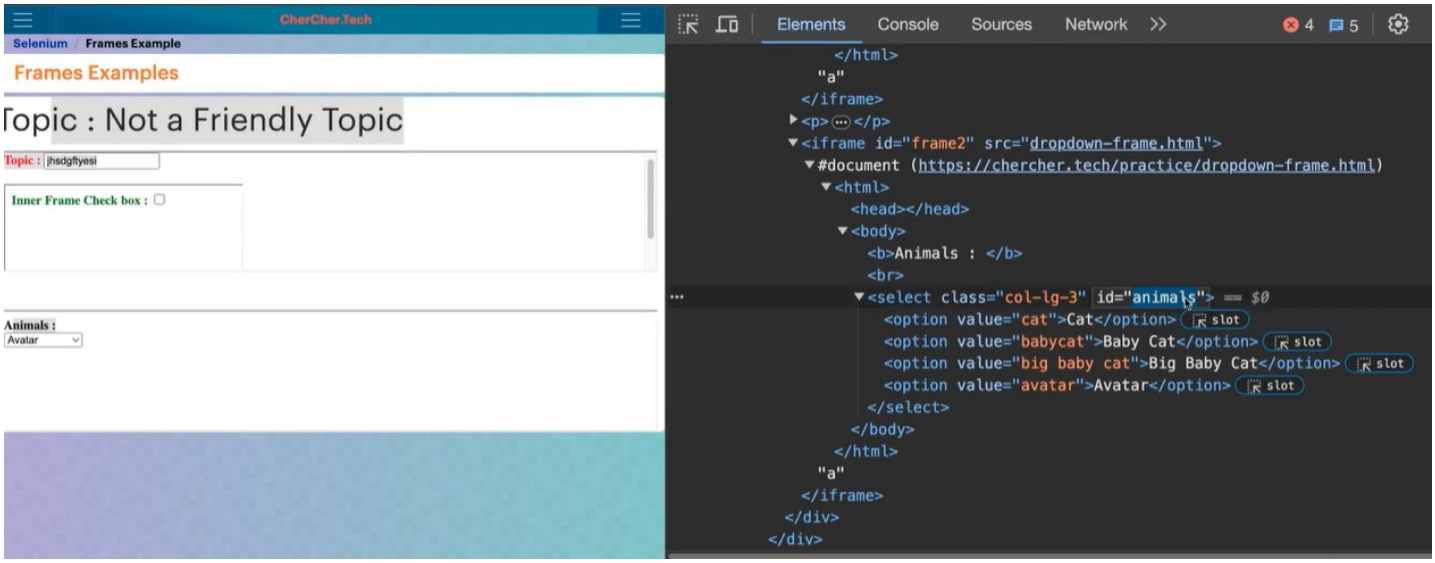
<ol class="breadcrumb nav">
flex
<li class="breadcrumb-item">
<li class="breadcrumb-item active update">
::after

flex

<ol class="breadcrumb h1">
<li class="breadcrumb-item">
<h1 class="breadcrumb-item">Frames Examples</h1> == $0

::after

ow
div.col-lg-8
ol.breadcrumb.h1
li.breadcrumb-item
h1.breadcrumb-item
h1.breadcrumb-item
1 of 1
Cancel




```
@Test(description = "Verify user can interact with web elements on the displayed")
    public void practiceNestedFrames() throws InterruptedException {
//Given user navigates to 'https://chercher.tech/practice/frames'
        driver.get("https://chercher.tech/practice/frames");
//Then user can see the page header "Frames Example"
        WebElement pageHeader = driver.findElement(By.cssSelector("h1.breadcrumb-item"));
        String expectedHeader= "Frames Example";
        String actualPageHeader = pageHeader.getText();
        Assert.assertEquals(actualPageHeader, expectedHeader, "The page header
verification failed!");
//And user can see "Topic: Not a Friendly Topic"
        WebElement topic = driver.findElement(By.tagName("Label"));
        String expectedTopic = "Topic: Not a Friendly Topic";
        String actualTopic  =  topic.getText();
        Assert.assertEquals(actualPageHeader, expectedHeader, "The page title
verification failed!");
//And user is able to type into Topic input box
        driver.switchTo().frame("frame1");
        WebElement topicInputBox =
driver.findElement(By.xpath("//*[@id='topic']/../input"));
        topicInputBox.sendKeys("Topic is iFrames!");
        Thread.sleep(5000);
//And user is able to check the "Inner Frame Check box"
        driver.switchTo().frame("frame3");
        WebElement checkBox = driver.findElement(By.id("a"));
        checkBox.click();
        Thread.sleep(5000);
//And user is able to select "Avatar" from Animals dropdown
// currently we are 2 levels down on iFrame 3
// in order to switch to another branch of three that is iframe2
// we need first to navigate to default content (main html page)
// and from there switchTo iFrame2
        driver.switchTo().defaultContent();
        driver.switchTo().frame("frame2");
        WebElement animalsDrop = driver.findElement(By.id("animals"));
```

```
        // With select dropdown we can simply send keys to the web element and it will
work

        //animalsDrop.sendKeys( "Avatar");

        // Thread.sleep(3000);
        String expectedSelection = "Avatar";
        Select animalsDropdown = new Select (animalsDrop);
        animalsDropdown.selectByVisibleText (expectedSelection);
        String actualSelectedOption=animalsDropdown.getFirstSelectedOption().getText();
        Assert.assertEquals (actualSelectedOption, expectedSelection,  "The dropdown
verification failed!");
    }
```

Explaination:

Let's break down the `@Test` method in the code and explain it step-by-step:

1. `@Test(description = "Verify user can interact with web elements on the displayed")`

- **`@Test` annotation**: Marks this method as a test method that can be executed by the TestNG framework.
- **`description` attribute**: Provides a description for the test method, which makes it easier to understand what the test is doing when reports are generated.

2. Test Method: `public void practiceNestedFrames()`

- This method contains the automation steps that interact with the web elements on the page to verify various functionalities of the web page located at `'https://chercher.tech/practice/frames'`.

3. Step-by-Step Breakdown:

a. Navigate to the webpage

```
driver.get("https://chercher.tech/practice/frames");
```

- This step opens the URL and loads the page in the browser.

b. Verify the page header

```
WebElement pageHeader = driver.findElement(By.cssSelector("h1.breadcrumb-item"));
String expectedHeader = "Frames Example";
String actualPageHeader = pageHeader.getText();
Assert.assertEquals(actualPageHeader, expectedHeader, "The page header verification
failed!");
```

- **Page Header Locator**: The `WebElement` for the header is found using a CSS selector for an `<h1>` tag with class `breadcrumb-item`.

- **Assertion**: The test verifies if the page header text matches `"Frames Example"`. If not, an assertion error will be thrown with the message `"The page header verification failed!"`.

c. Verify the Topic Label

```
WebElement topic = driver.findElement(By.tagName("Label"));
String expectedTopic = "Topic: Not a Friendly Topic";
String actualTopic = topic.getText();
Assert.assertEquals(actualTopic, expectedTopic, "The page title verification failed!");
```

- **Topic Label Locator**: The `WebElement` for the label is found by its tag name `<label>`.
- **Assertion**: This step checks if the label text is `"Topic: Not a Friendly Topic"`. If it doesn't match, it throws an assertion failure with the message `"The page title verification failed!"`.

d. Interact with the input box in `frame1`

```
driver.switchTo().frame("frame1");
WebElement topicInputBox = driver.findElement(By.xpath("//*[@id='topic']/../input"));
topicInputBox.sendKeys("Topic is iFrames!");
Thread.sleep(5000);
```

- **Switch to Frame**: Before interacting with the input box, the test switches to the first iframe (`frame1`) because the input element is inside it.
- **Interact with Input Box**: It locates the input box using XPath and sends the text `"Topic is iFrames!"`.

e. Check the checkbox inside `frame3` (nested frame)

```
driver.switchTo().frame("frame3");
WebElement checkBox = driver.findElement(By.id("a"));
checkBox.click();
Thread.sleep(5000);
```

- **Nested Frame**: The test navigates further into `frame3`, which is inside `frame1`.
- **Interact with Checkbox**: After switching to `frame3`, the test finds the checkbox element by its ID and clicks it.

f. Select an option from the dropdown in `frame2`

```
driver.switchTo().defaultContent();
driver.switchTo().frame("frame2");
WebElement animalsDrop = driver.findElement(By.id("animals"));
Select animalsDropdown = new Select(animalsDrop);
animalsDropdown.selectByVisibleText("Avatar");
String actualSelectedOption = animalsDropdown.getFirstSelectedOption().getText();
Assert.assertEquals(actualSelectedOption, "Avatar", "The dropdown verification failed!");
```

- **Switching Back to Default Content**: Since the dropdown is located in a different frame (`frame2`), the test first switches back to the default (main) content using `driver.switchTo().defaultContent()`.
- **Switch to `frame2`**: After returning to the main page, it then switches to `frame2` to interact with the dropdown.
- **Select from Dropdown**: The test selects the option `"Avatar"` from the dropdown using `selectByVisibleText()`.
- **Assertion**: Finally, it checks whether the selected option is `"Avatar"` and asserts it. If not, an assertion failure message `"The dropdown verification failed!"` is displayed.

Key Points:

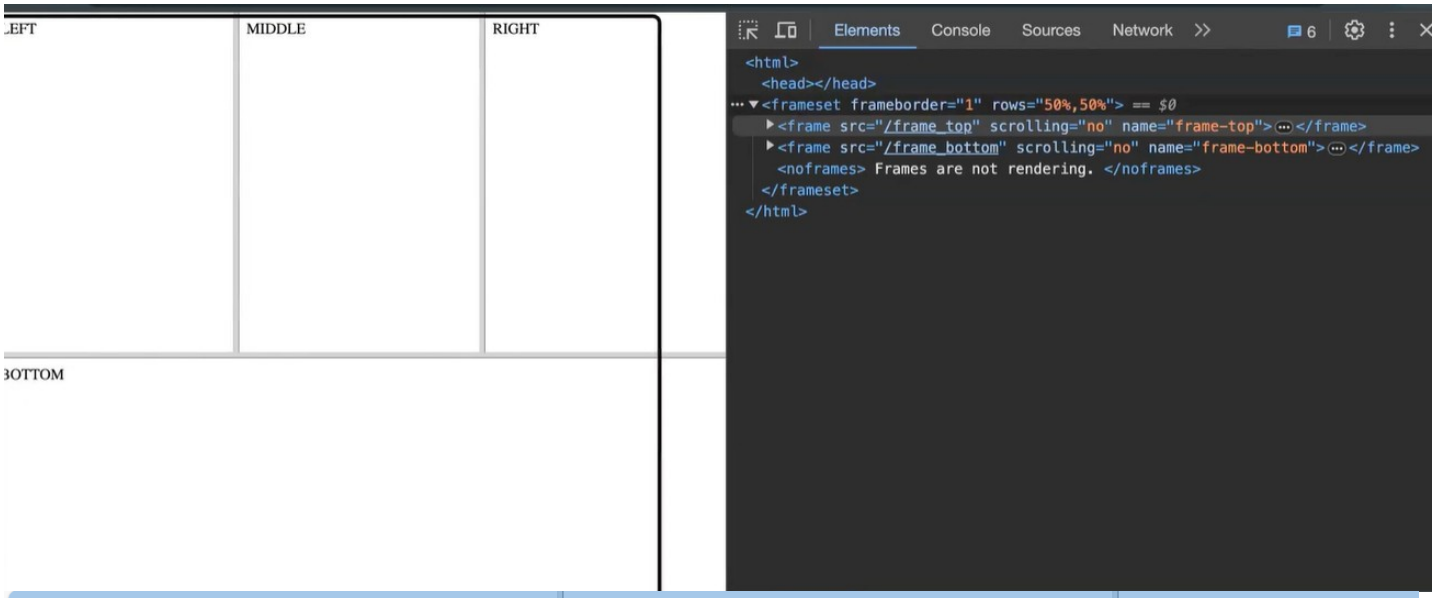
- **Frame Handling**: The test switches between different frames on the page using `driver.switchTo().frame()`. If you're inside one frame and need to interact with another, you first switch back to the default content and then move to the target frame.
- **Assertions**: The test uses `Assert.assertEquals()` to verify that the actual web elements' values match the expected values. This is crucial for validating that the page behaves as expected.
- **Select Dropdown**: The test uses Selenium's `Select` class to handle dropdowns, which allows selecting options by visible text, index, or value.

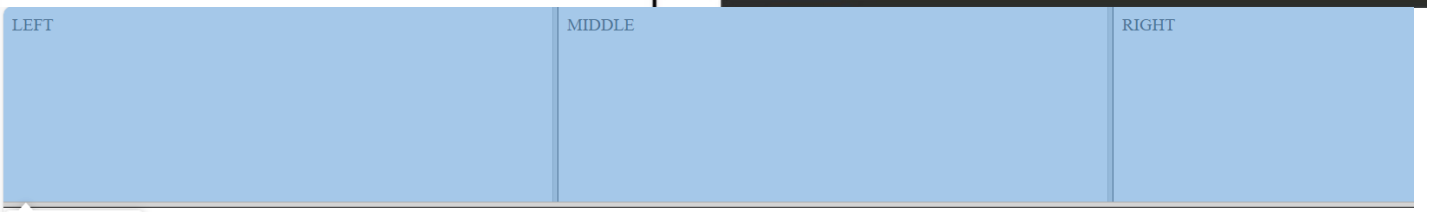
Overall Flow:

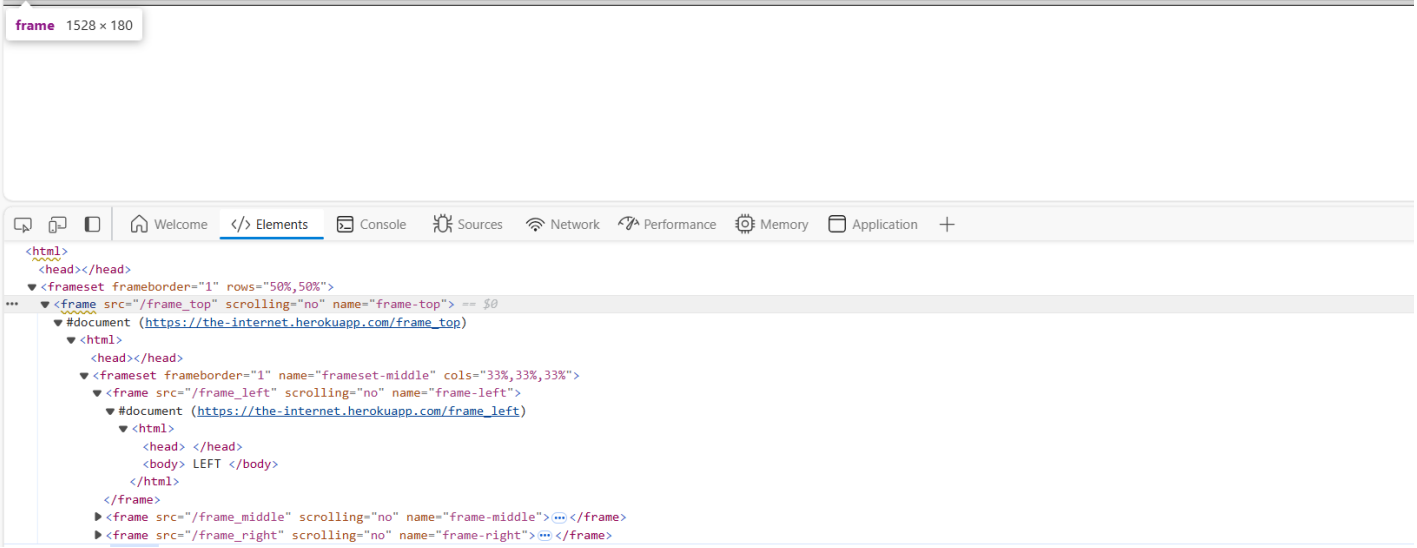
1. Load the webpage.
2. Verify the presence of the correct header and topic.
3. Switch between nested iframes to interact with different elements (text box, checkbox, dropdown).
4. Validate each interaction with assertions.

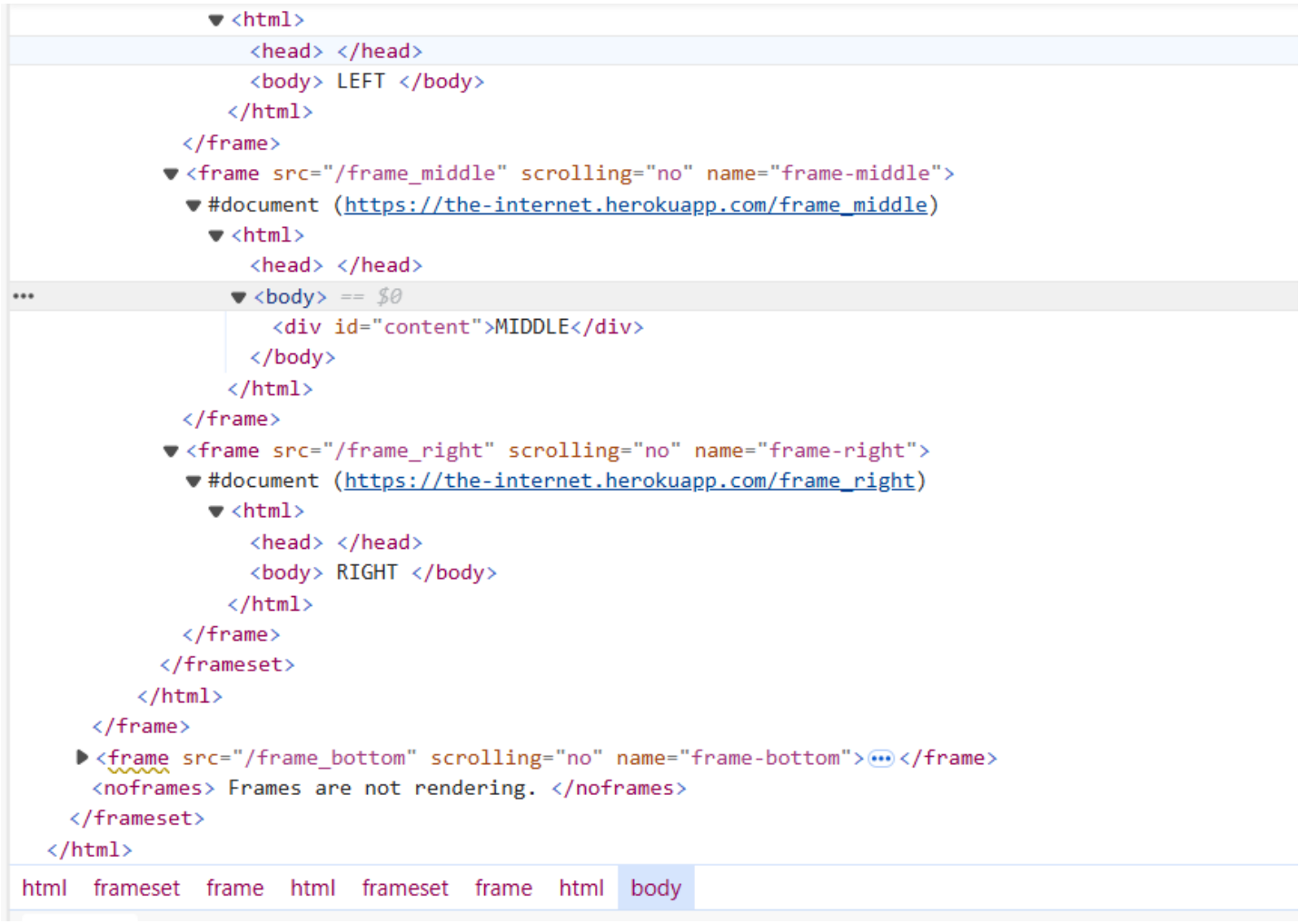
Scenario: Verify user can see all frames content textGiven user lands to http://the-internet.herokuapp.com/
When user clicks on "Frames" tab
And user clicks on "Nested Frames" tab
Then user is able to see LEFT, RIGHT, MIDDLE and BOTTOM texts are displayed in page.

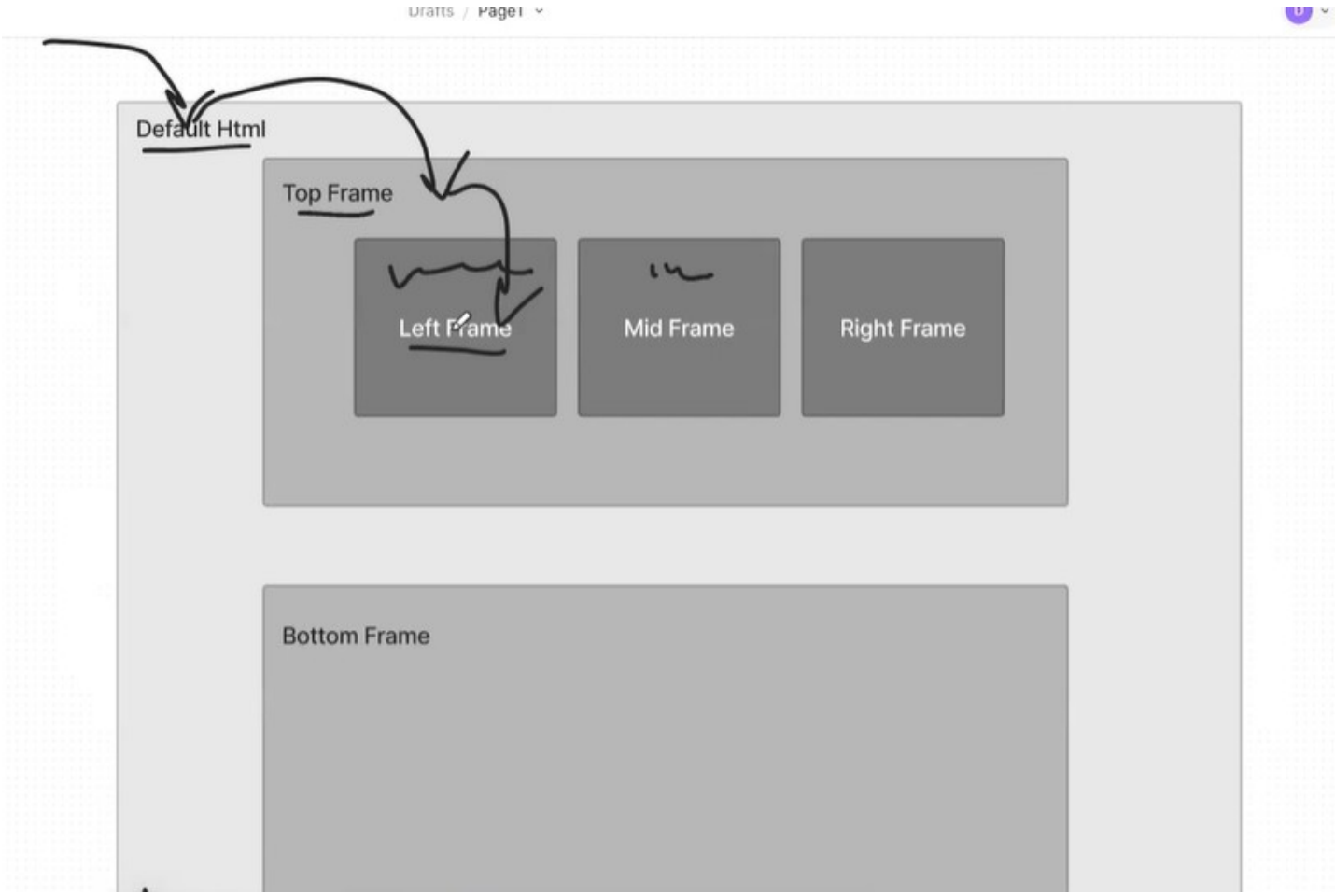
We have top and bottom


LEFT
MIDDLE
RIGHT
BOTTOM
Elements
Console
Sources
Network
6
<html>
<head></head>
<frameset frameborder="1" rows="50%,50%"> == $0
<frame src="/frame_top" scrolling="no" name="frame-top">…</frame>
<frame src="/frame_bottom" scrolling="no" name="frame-bottom">…</frame>
<noframes> Frames are not rendering. </noframes>
</frameset>
</html>


LEFT
MIDDLE
RIGHT


frame 1528 × 180
Welcome
Elements
Console
Sources
Network
Performance
Memory
Application
<html>
<head></head>
<frameset frameborder="1" rows="50%,50%">
<frame src="/frame_top" scrolling="no" name="frame-top"> == $0
#document (https://the-internet.herokuapp.com/frame_top)
<html>
<head></head>
<frameset frameborder="1" name="frameset-middle" cols="33%,33%,33%">
<frame src="/frame_left" scrolling="no" name="frame-left">
#document (https://the-internet.herokuapp.com/frame_left)
<html>
<head> </head>
<body> LEFT </body>
</html>
</frame>
<frame src="/frame_middle" scrolling="no" name="frame-middle">…</frame>
<frame src="/frame_right" scrolling="no" name="frame-right">…</frame>


▼ <html>
<head> </head>
<body> LEFT </body>
</html>
</frame>
▼ <frame src="/frame_middle" scrolling="no" name="frame-middle">
▼ #document (https://the-internet.herokuapp.com/frame_middle)
▼ <html>
<head> </head>
▼ <body> == $0
MIDDLE
</body>
</html>
</frame>
▼ <frame src="/frame_right" scrolling="no" name="frame-right">
▼ #document (https://the-internet.herokuapp.com/frame_right)
▼ <html>
<head> </head>
<body> RIGHT </body>
</html>
</frame>
</frameset>
</html>
</frame>
▶ <frame src="/frame_bottom" scrolling="no" name="frame-bottom"> ... </frame>
<noframes> Frames are not rendering. </noframes>
</frameset>
</html>
html frameset frame html frameset frame html body




```
@Test
    public void practiceFrame_setTopBottom(){
//Given user lands to 'http://the-internet.herokuapp.com/'
        driver.get("http://the-internet.herokuapp.com/");
//When user clicks on "Frames" tab
        WebElement framesTab = driver.findElement(By.xpath(  "//*[text() = 'Frames']"));
        framesTab.click();
//When user clicks on "Nested Frames" tab
        WebElement nestedFramesTab = driver.findElement(By.xpath(  "//*[text() = 'Nested
Frames']"));
        nestedFramesTab.click();
// Get the value of the left frame on the page.
        driver.switchTo().frame(  "frame-top");
        driver.switchTo().frame( "frame-left");
        String expectedContentLeftFrame = "LEFT";
        String actualContentLeftFrame = driver.findElement(By.tagName
("body")).getText();
        Assert.assertEquals (actualContentLeftFrame, expectedContentLeftFrame,  "The Left
Frame content verification failed!");
    }
```

Explaibation:

Let's break down the `@Test` method `practiceFrame_setTopBottom()` step by step to understand its purpose, logic, and flow.

1. `@Test`

- Marks this method as a test case that will be executed by the TestNG framework.

2. Test Method: `public void practiceFrame_setTopBottom()`

This method tests the functionality of switching between nested frames on a webpage and verifying the content in a specific frame. It simulates a user interacting with the frames on the page located at `'http://the-internet.herokuapp.com/'`.

Step-by-Step Breakdown:

a. Navigate to the webpage

```
driver.get("http://the-internet.herokuapp.com/");
```

- **`driver.get()`**: This opens the provided URL in the browser. In this case, it loads the webpage located at `http://the-internet.herokuapp.com/`, which contains various test scenarios, including frames.

b. Click on the "Frames" tab

```
WebElement framesTab = driver.findElement(By.xpath("//*[text() = 'Frames']"));
framesTab.click();
```

- **Locate the "Frames" Tab**: The test finds the "Frames" tab by locating the element that contains the text `"Frames"`. This is done using an XPath expression that matches any element with the given text.
- **Click on the Tab**: After locating the tab, it simulates a click action to navigate to the "Frames" section of the website, which contains examples of different frame setups.

c. Click on the "Nested Frames" tab

```
WebElement nestedFramesTab = driver.findElement(By.xpath("//*[text() = 'Nested
Frames']"));
nestedFramesTab.click();
```

- **Locate the "Nested Frames" Tab**: Once the "Frames" page is displayed, the test finds the "Nested Frames" option by its text using another XPath expression.
- **Click on the Tab**: The test simulates a click on the "Nested Frames" tab, which leads to a page containing multiple nested frames (frames within frames).

d. Switch to the "frame-top"

```
driver.switchTo().frame("frame-top");
```

- **Switch to the Parent Frame**: The test switches to the `frame-top` frame using `driver.switchTo().frame()`. This frame is the parent frame that contains three child frames: `frame-left`, `frame-middle`, and `frame-right`. In this step, the browser's focus is now inside the `frame-top` frame, so any further actions will target its content.

e. Switch to the "frame-left"

```
driver.switchTo().frame("frame-left");
```

- **Nested Frame**: After switching to the parent frame (`frame-top`), the test further switches to the `frame-left` frame, which is one of the child frames inside `frame-top`. The test must switch to this frame to access and interact with its content (i.e., the text it contains).

f. Get and Verify the Content of the Left Frame

```
String expectedContentLeftFrame = "LEFT";
String actualContentLeftFrame = driver.findElement(By.tagName("body")).getText();
Assert.assertEquals(actualContentLeftFrame, expectedContentLeftFrame, "The Left Frame
content verification failed!");
```

- **Locate the Content**: After switching to `frame-left`, the test locates the text content inside the body tag (`<body>`) of the frame.
- **Expected Content**: The expected text inside the left frame is `"LEFT"`.
- **Assertion**: The test checks if the actual text found in `frame-left` matches the expected text (`"LEFT"`). If it doesn't match, an assertion failure will occur, and the message `"The Left Frame content verification failed!"` will be shown.

Summary of the Test:

- **Navigating Through the Web Page**: The test navigates through different elements of the webpage: from the main page to the "Frames" section, then to the "Nested Frames" example.
- **Switching Between Frames**:
  - **Step 1**: Switch to the parent frame (`frame-top`), which contains three child frames.
  - **Step 2**: Switch to the `frame-left` (a nested frame inside `frame-top`).
- **Content Verification**:
  - After switching to `frame-left`, the test retrieves the text content inside this frame and verifies it against the expected text (`"LEFT"`).
  - If the content does not match, the test fails with an assertion error.

Key Concepts:

1. **Frame Switching**:
   - In Selenium, you must switch to the correct frame using `driver.switchTo().frame()` before interacting with elements inside that frame.
   - Frames are often used to embed independent content, and the test must handle switching between them appropriately.
2. **Nested Frames**:
   - When dealing with nested frames (frames within frames), you first switch to the parent frame, and then further switch to the child frame. Here, the test first switches to `frame-top` and then to `frame-left`.
3. **Assertions**:
   - `Assert.assertEquals()` is used to verify that the actual content of the left frame matches the expected content. If it doesn't, the test will fail with the provided message.

Overall Flow:

1. Load the webpage.
2. Navigate to the "Frames" section.
3. Navigate to the "Nested Frames" example.
4. Switch to the `frame-top` and then to the `frame-left` frame.
5. Retrieve and verify the text content inside the left frame.